Parallel Crystal
Version 2.4
Developer's Manual

The information given in this manual refers to Version 2.4 of Parallel Crystal by Dynalivery Corp.

Dynalivery Corp. endeavors to ensure that the information in this manual is correct, but does not accept liability for any error or omission.

The development of Dynalivery Corp. products and services is continuous and published information may not be up-to-date. Any particular issue of a product may contain part only of the facilities described in this manual, or may contain facilities not described here.

Specifications and statements relating to performance in this manual are Dynalivery Corp. estimates intended for general guidance. They may require adjustment in particular circumstances and are therefore not formal offers or undertakings.

Statements in this manual are not part of a contract or program product license save insofar as they are incorporated into a contract or license by express reference. Issue of this manual does not entitle the recipient access to or use of the products described, and such access or use may be subject to separate contracts or licenses.

Copyright

This manual is copyrighted, with all rights reserved. Under copyright laws, this manual may not be copied, in whole or in part, without the written permission of Dynalivery Corp.

Copyright © 2000 Dynalivery Corp.

Trademarks

Parallel Crystal is a trademark of Dynalivery Corp.

Seagate Crystal Reports is a trademark of Seagate Software, Inc. Visibroker is a trademark of Inprise Corporation. Other brand or product names are trademarks or registered trademarks of their respective holders.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Dynalivery Corp.
320 West Port Plaza, Suite 205,
St Louis, Missouri, 63146 USA

Web http://www.dynalivery.com/
Tel. (314) 205-8995
Fax. (314) 205-8622

Table of Contents

Chapter 1 Introducing Parallel Crystal....9

Using this Manual....10
RoadMap....10
Things Not Included in this Manual....11

Getting Help....12
Using our Web Site....12
Email Tech Support....12
Telephone Support....13
Getting Software Updates....13

Why use Parallel Crystal?....14
Report Basics....14
Customizing Reports....15
Parallel Crystal Report Engine....16
Advantages of Using a Report Server....18
Distributed Report Server Processing....21
Parallel Crystal and Web Application Servers....21
Parallel Crystal Client and Server Components....24
Parallel Crystal PDF Library....27

Chapter 2 Getting Started with Java Clients....29

Using Parallel Crystal in a Java Application....30
Generating a Simple PostScript Report....32
Trouble Shooting the Simple Client....35
Generating a Simple PDF Report....37
Generating a Simple HTML Report....38
Summary....40

Using Parallel Crystal in a Java Applet....41
<Applet> Tag Details....48
Remarks on Applet Programming....49

Chapter 3 Using the Java Client Package....51

Parallel Crystal Clients....53
Services for Clients....53
Application Client Instantiation....54
Applet Client Instantiation....55
Client Configuration....56
PCREGateway Properties....57
PCREServer Properties....58
Application Client Connection....60
Applet Client Connection....61
Client ORB Management....65
Client Disconnection....66

Report Generation *67*
Accessing Report Engines and Print Jobs 67
Customizing Print Jobs 69
Print Job Execution 69
Print Job Details 70
Print Job Outputs 70
Formula Manipulation 75
Parameter Field and Stored Procedure Manipulation 75
Database Table Manipulation 76
SQL Queries 77
Sort Field Manipulation 77
Sections and Groups 78
Area and Section Formatting 78
Sub report Management 79
Graph Manipulation 79

Data Object Reporting *81*
Overview 81
Preparation 81
Analyze the Data Source 82
DOR Design Utility 85
DOR to RPT Utility 85
Final Report Design 86
Running a DOR Report 87
Tips for Using DOR 88

Error Recovery *92*

Console Interface *96*

Parallel Crystal Report Retrieval *99*
Retrieval to Application Clients 99
Retrieval to Applet Clients 100

Parallel Crystal Configuration Server *101*

Parallel Crystal Load Balancer *103*
Automatic Mode 103
Manual Mode 103

Multi-threaded Clients *106*

Chapter 4 Trouble Shooting Java Clients 109

Problems with CLASSPATH 109
Common Problems with Client Connections 110
Problems with Applets 112
Problems with Export Formats 114
Problems with PDF 114
Problems with the Report Server 116

Chapter 5 Getting Started with ActiveX Clients 119

Parallel Crystal and ActiveX/COM clients *120*
Microsoft COM 120
Microsoft COM Automation 122
The Parallel Crystal Object Model 124
The Parallel Crystal Automation Server 126

Using Parallel Crystal with Visual C++ *127*
Generating PostScript Reports with COM 127
Using Simple Wrapper Classes 131
Generating PDF and HTML Reports with COM 133
Generating PostScript Reports with Visual C++ and COM 134
Generating PDF and HTML Reports with VC++ and COM 140
The Trouble with Smart Pointers 141
Handling Errors from COM 142
Trouble Shooting Simple C++ COM Clients 143
Things to Remember 144

Using Parallel Crystal with Visual J++ *145*
Visual J++ Preliminaries 145
Generating PostScript Reports with Visual J++ 147
Generating PDF and HTML Reports with Visual J++ 149
Handling Errors from Visual J++ 150
Trouble Shooting Simple Visual J++ Clients 151

Using Parallel Crystal with Visual Basic *153*
Visual Basic Preliminaries 153
Generating PostScript Reports with Visual Basic 154
Generating PDF and HTML Reports with Visual Basic 158
Handling Errors from Visual Basic 159
Trouble Shooting Simple Visual Basic Clients 160

Using Parallel Crystal with Active Server Pages *162*
ASP Preliminaries 165
Generating HTML Reports with ASP 166
Retrieving Reports with ASP 173
Generating PDF and PostScript Reports with ASP 175
Handling Errors from ASP 2.0 176
Trouble Shooting Simple ASP Clients 177

Chapter 6 Using the PCRE Automation Server 179

Automation Server Clients *180*
Connecting to the Automation Server 180
Connecting to the Report Server 183
Disconnecting from the Report Server 184
Disconnecting from the Automation Server 185

Report Generation *186*
ICRApplication 189
ICRGlobalOptions 190
ICRServer 190
ICRReport 191
ICRArea(s) 192
ICRAreaOptions 193
ICRGroupAreaOptions 194
ICRSection(s) 194
ICRSectionOptions 196
ICRReportObject(s) 197
ICRSortField(s) 198
ICRDataBase 199
ICRDataBaseTable(s) 199
ICRDataBaseFieldDefinition(s) 201
ICRDataBaseParameter(s) 202
ICRPrintingStatus 203
ICRPrinterInfo 204
ICRReportSummaryInfo 205
ICRExportOptions 206
ICRReportOptions 207
ICRFormulaFieldDefinition(s) 208
ICRParameterFieldDefinition(s) 209

PDF Report Generation *211*

Parallel Crystal Configuration Server *212*

Parallel Crystal Report Retrieval *215*

Parallel Crystal Load Balancer *217*
Using the Load Balancer in Automatic Mode 217
Using the Load Balancer in Manual Mode 218
Using Manual Load Balancing with Visual C++ 220
Using Manual Load Balancing with Visual Basic 221
Using Manual Load Balancing with Visual J++ 223

Chapter 7 Trouble Shooting ActiveX Clients 225

Problems with Client Connections 225
Problems with Export Formats 227
Problems with PDF 228
Problems with the Report Server 229

Chapter 8 Getting Started with C/C++ Clients 231

Using Parallel Crystal in a C++ Application 234
Generating a Simple PostScript Report 234
Trouble Shooting the Simple C++ Client 237
Generating a Simple PDF Report 238
Generating a Simple HTML Report 239
Summary 240

Using Parallel Crystal in a C Application 241
Generating a Simple PostScript Report 241
Trouble Shooting the Simple C Client 243
Generating a Simple PDF Report 245
Generating a Simple HTML Report 246

Chapter 9 Using the C/C++ Client Library 249

Parallel Crystal Clients 251
Services for Clients 251
C++ Application Client Instantiation 252
ANSI C Application Client Initialization 253
C++ Client Configuration 255
Accessing PCREGateway Properties from C++ 257
Accessing PCREServer Properties from C++ 258
C Client Configuration 259
Accessing PCREGateway Properties from C 261
Accessing PCREServer Properties from C 263
Connecting C++ Clients 265
Disconnecting C++ Clients 267
Connecting C Clients 269
Disconnecting C Clients 271

Report Generation for C++ Clients 272
Accessing Report Engines and Print Jobs 272
Customizing Print Jobs 274
Print Job Execution 275
Print Job Details 275
Print Job Outputs 275
Formula Manipulation 281
Parameter Field and Stored Procedure Manipulation 281
DataBase Table Manipulation 282
SQL Queries 282
Sort Field Manipulation 283
Sections and Groups 283
Area and Section Formatting 284
Subreport Management 284
Graph Manipulation 285

Report Generation for C Clients 286

Error Recovery in C Clients 288

Error Recovery in C++ Clients 289

Console Interface 293

Report Retrieval 295
Retrieval to C++ Clients 295
Retrieval to C Clients 296

Parallel Crystal Configuration Server *298*
Configuration Server for C++ Clients 298
Configuration Server for C Clients 300

Parallel Crystal Load Balancer *302*
Using the Load Balancer from C++ 302
Using the Load Balancer from C 304

Multi-Threaded Clients *305*

Chapter 10 Trouble Shooting C/C++ Clients 309

Problems with Client Connections 309
Problems with Export Formats 311
Problems with PDF 311
Problems with the Report Server 313

Appendix A Simple ASP/VBScript Tutorial 315

Overview 315
Understanding a Simple Program 317
Embedding the Program in a Working ASP Page 320
Setting Up Your First ASP Application 322
In the Event of a Failure 323
Diagnosing Server-side Setup Problems 324
Diagnosing Client (Web Server)-side Setup Problems 324

Chapter 1
Introducing Parallel Crystal

This manual describes how to use Dynalivery's Parallel Crystal Report Server to customize and generate reports prepared with the Seagate Crystal Reports Designer.

The introductory chapter is divided into three sections

- The section entitled Using this Manual describes the layout and composition of this manual, including a Roadmap to help locate the information you need.

- The section entitled Getting Help describes how to get technical assistance, how you should report bugs, and how you can get the latest versions of Parallel Crystal from our web site.

- The section entitled Why Use Parallel Crystal? includes a detailed description of the design and operation of the Parallel Crystal product, as well as an explanation of some basic report processing concepts.

Note: It is highly recommended that you read most of this introductory chapter. It explains many of the basic concepts of the design and operation of the Parallel Crystal product.

Using this Manual

This User Manual is intended to get you started with Parallel Crystal, in particular to get that all-important first report produced. This manual is essentially divided into three parts, or paths, that separately describe how you may use Parallel Crystal. Use the following RoadMap to navigate to the correct path in the manual.

RoadMap

Your route through the roadmap depends upon your client development environment or on a prior selection of web technology. Try to match your requirements against the following queries.

- Are you using a "pure Java" client development environment such as the Sun JDK, or are you using a Java-oriented Web Application Server? Or are you using Visual J++ but without the ActiveX/COM extensions? If so,

  Go to Chapter 2 entitled Getting Started with Java Clients to get started.

  Go to Chapter 3 entitled Using the Java Client Package to learn more about using the Parallel Crystal Java Client package to write client applications and applets.

  Go to Chapter 4 entitled Trouble Shooting Java Clients to learn how to solve the problems that occur most often in Java clients.

- Are you using any of the following Microsoft products for development: Active Server Pages with VBScript or Jscript, or Visual Basic? Or do you wish to use the ActiveX/COM extensions provided in Visual C++ or Visual J++? If so,

  Go to Chapter 5 entitled Getting Started with ActiveX Clients to get started.

  Go to Chapter 6 entitled Using the PCRE Automation Server to learn more about the customization interface provided by the PCRE Automation Server.

  Go to Chapter 7 entitled Trouble Shooting ActiveX Clients to learn how to solve the problems that occur most often with ActiveX clients.

- Are you using a C or C++ client development environment? Or are you using Visual C++ but without the ActiveX/COM extensions? If so,

  Go to Chapter 8 entitled Getting Started with C/C++ Clients to get started.

  Go to Chapter 9 entitled Using the C/C++ Client Library to learn how to use the Parallel Crystal C/C++ Client Library.

  Go to Chapter 10 entitled Trouble Shooting C/C++ Clients to learn how to solve the problems that occur most often with C/C++ clients.

Things Not Included in this Manual

If the Roadmap did not help you locate an appropriate chapter, maybe the facilities you require are not currently available in this volume.

Information this manual does not contain

- Installation instructions.
- System Administration information and trouble-shooting.
- Networking information and trouble-shooting.

*Note: The **Parallel Crystal System Administrator's Manual** is being designed and developed to describe these items.*

PRCE Customization APIs

If you're looking for detailed descriptions of the PCRE Customization APIs, you should consult the online *Parallel Crystal API Reference Documentation.* The API Reference contains detailed descriptions of all the API classes and methods that comprise the various PCRE client packages. Whenever you get an API programming error, you should go straight to the Reference Documentation to get help. The *Parallel Crystal API Reference Documentation* is automatically installed with Parallel Crystal and is available from the Start/Programs/Parallel Crystal/Documentation option on the Windows Start Menu.

Getting Help

Dynalivery Corp. provides a variety of sources of information for problem solving. We would invite you to visit our website, or email us with questions, comments, suggestions, and problems.

Using our Web Site

Our web site contains links to the latest online versions of this documentation and many other helpful resources.

http://www.dynalivery.com

Email Tech Support

Technical Support for products of Dynalivery Corp. starts with an email to:

TechSupport@dynalivery.com.

In your email to Tech Support, please include

- A copy of the Report File (*filename*.RPT).
- A copy of the Server Log file (server.log is usually found in the \\MobileApps\Pcre\logs folder).
- A small sample of the code being used to call the execution of the report file.
- A detailed description of the problem.
- A detailed description of the configuration environment being used, including:
    - The Parallel Crystal version number.
    - Operating System.
    - Web Application Server (if applicable)
    - The client development package: ASP, Visual Basic, Visual J++, Visual C++, pure Java, C, C++, etc.
- A sample of some of the data being used (if applicable).
- Information about any stored procedures being used in the report file.

Telephone Support

We offer telephone support at the following numbers

USA	1-888-662-7771
	314-205-8995

We would like to encourage you to use our Tech Support email address before calling on the telephone for support. We are always happy to take your call, but generally, we have found that we have better and more accurate responses to inquiries when they are handled through email. Thanks.

Getting Software Updates

Updates to the Parallel Crystal software are available on our internet site at http://www.dynalivery.com. Select 'downloads'. To get permission to download, you use a user name derived from your Parallel Crystal Product Serial Number, and a password that is included on the back cover of your Parallel Crystal CD. For example, if your Serial Number is 9807-004, then your user name is 9807004. Your FTP account is valid for the period of your evaluation, and for one full year thereafter, if you purchase the software.

Why use Parallel Crystal?

Parallel Crystal incorporates an enhanced version of Seagate Software's popular Crystal Report Engine into a multi-user server environment. You can connect remotely to the server and perform report customization and generation tasks using the standard Crystal Report APIs and a variety of client environments.

Parallel Crystal may be used in a variety of client environments, including

- Microsoft ActiveX products such as Active Server Pages and Visual Basic.
- "Pure" Java applications and applets.
- Conventional C or C++ applications.
- Parallel Crystal may also be accessed from a number of Java-based Web Application Servers.

In the sections that follow, we'll describe some basic concepts about report creation and customization. We'll also explain why it is productive to migrate the process from a desktop computer to a more powerful, networked Report Server.

Report Basics

A report that is designed with Crystal Report Designer is created as a template that is stored in a report file. The template holds all the information necessary to integrate selected data records, perform appropriate calculations and integrate the results into a formatted document. Report files are usually identified with an `.rpt` suffix.

Normally, data is first added to a report as part of a print-preview process that allows a mock-up of the final report to be viewed within an inspection window. If the results are satisfactory, then either the template, or the template and data, may be re-saved in the report file. If the first option is chosen, the report is said to possess "saved data". Report pre-viewing is illustrated in Figure 1.1.

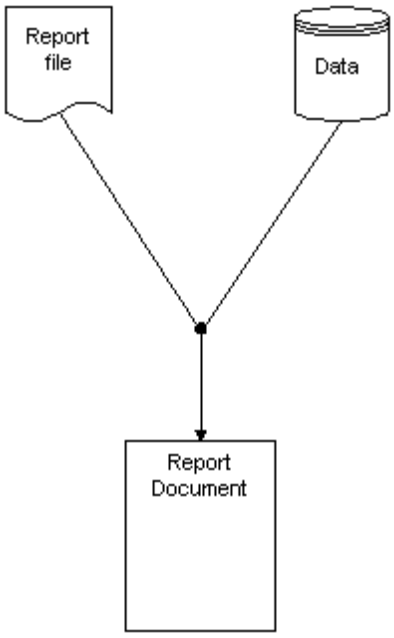


Figure 1.1

Customizing Reports

The report files produced by the Crystal Report Designer support an additional modification process called "customization" which allows many of the design time properties of the report to be adjusted at runtime, just prior to report generation. Customization is performed by a software component called the Crystal Reports Print Engine that reads the original report file, applies modifications to the report properties and then merges the data to produce the final report. Customization is frequently used to modify the selection criteria that are used to filter data records from a database or to specify the database at runtime. Report customization is illustrated in Figure 1.2.

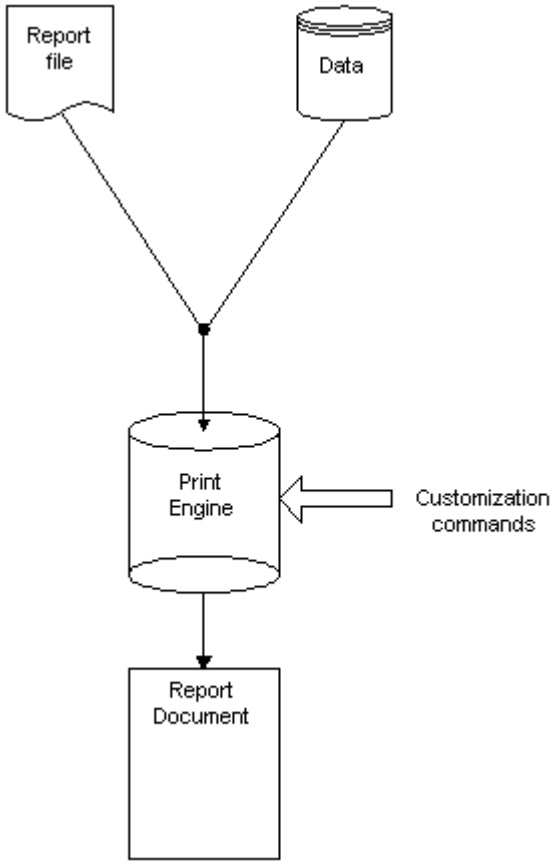


Figure 1.2

Parallel Crystal Report Engine

The Crystal Report Print Engine is provided as a Dynamic Link Library (DLL) that accepts customization “commands” delivered as procedure calls across an API. In order to perform customization you must write a program or script that calls the appropriate API functions in the DLL.

The Parallel Crystal Report Server provides report customization through a licensed version of the Crystal DLL called the Parallel Crystal Report Engine (PCRE). The PCRE is an extension and enhancement for the Crystal Report Print Engine.

PCRE extensions and enhancements include

- Provide access to the customization API for the Java programming language.
- Enable reports to be produced directly in Adobe's Portable Document Format (PDF).
- Enable access to the customization API for networked clients.
- Provide an adaptation of the API for web-related technologies such as Microsoft Active Server Pages (ASP).
- Secure performance enhancements on multi-processor architectures.

The customization API provided by the Report Engine contains over a hundred functions that provide management facilities for print jobs, database access, report formats and report destinations, report layout, data record selection, formula manipulation, and error reporting.

The API is expressed in the following language-related ways

- As a set of ANSI C function declarations that can be called from a C program or any other programming language that supports C-style function linkage.
- As a set of packaged Java classes which can be invoked from Java applications or applets.
- As a C++ class library which can be called from C++ applications.
- As a Microsoft COM Automation Server which can be accessed from COM-enabled clients such as Visual Basic, Visual Basic Script, JScript, Visual C++ and Visual J++.

If you are a C programmer or are familiar with the original Seagate Crystal Report Engine, you will find everything you need in the ANSI C API. If your preference is for object-oriented languages like C++ or Java, then it is more likely that you will want to use the simple C++ class library or the Java package. Finally, if you are familiar with Visual Basic or Microsoft Active Server Pages, then the scripting interface offered by the Automation Server will most likely be the simplest option.
In order to convey the general flavor of working with the PCRE customization API, Example 1.1 shows a simple Java program that generates a report using one of the well-known sample Crystal report files. Even if you don't know the Java programming language, the example should convey the notion of connecting to a print engine, specifying the details of a print job that is to run on that engine, and finally starting the job. The final report will be sent to an available printer.

Parallel Crystal integrates the various versions of the customization API into client frameworks that allow Report Engines to be accessed on networked Report Servers. In the next section we explain the Report Server concept in greater detail and describe how the client frameworks may be used within the context of the modern networked environments.

```
import com.mobileapps.pcre.*;

class Simple
{
   public static void main( String [] args )
   {
      PCREApplicationClient client = null;
      try
      {
         client = new PCREApplicationClient(“example.host.com);
         client.Connect();
         PCREEngine engine = client.OpenEngine();
         PCREJob job = engine.OpenJob("Box.rpt");
         job.OutputToPrinter(1);
         job.Start();
         job.Close();
         engine.Close();
      }
      catch ( PCREError ex )
      {
        System.out.println(“Exception: “+ex);
      }
      finally
      {
        if (client != null) client.Quit();
      }
   }
}
```

Example 1.1

Advantages of Using a Report Server

Crystal Reports supports report customization by allowing desktop application programs to be linked to the Print Engine DLL. In most cases, the application is written in Visual Basic and uses a version of the customization API distributed with the Visual Basic environment. If the report requires access to data, the database is located either in the same desktop machine, or in a networked database server. For example, a typical arrangement involving a SQL DBMS is shown in Figure 1.3.

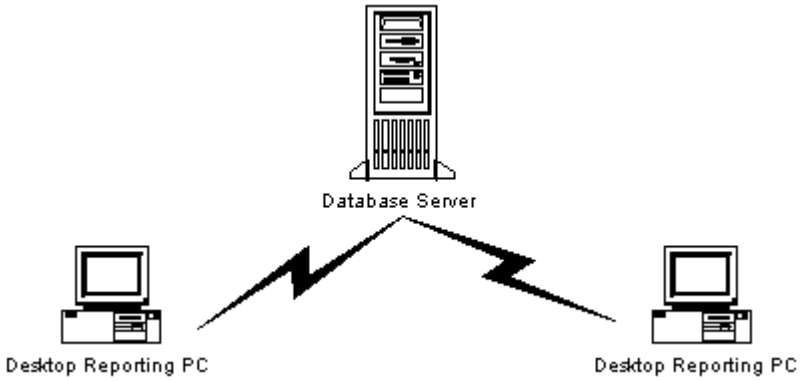


Figure 1.3

Each PC runs the customization application linked with the Crystal Print Engine DLL. However executing an appropriate SQL query in the database server performs the selection of data records. The records are returned to the PCs for incorporation into the report. An application in which clients communicate with a single server is sometimes referred to as having a *two-tier* client-server architecture.

While there are many advantages in the two-tier arrangement, each client must have its own copy of the original report file and must run its own copy of the customizing application. When an application such as this is deployed on a largescale the cost of duplicating the reporting capability in each client becomes significant. Within a corporate environment it may also be difficult or even impossible to deploy multiple copies of report files to each client PC, or to schedule regular report runs.

Parallel Crystal solves many of the problems associated with two-tier systems by moving the report files and the Print Engine DLL to a Report Server machine. This arrangement is shown in Figure 1.4 on page 20.

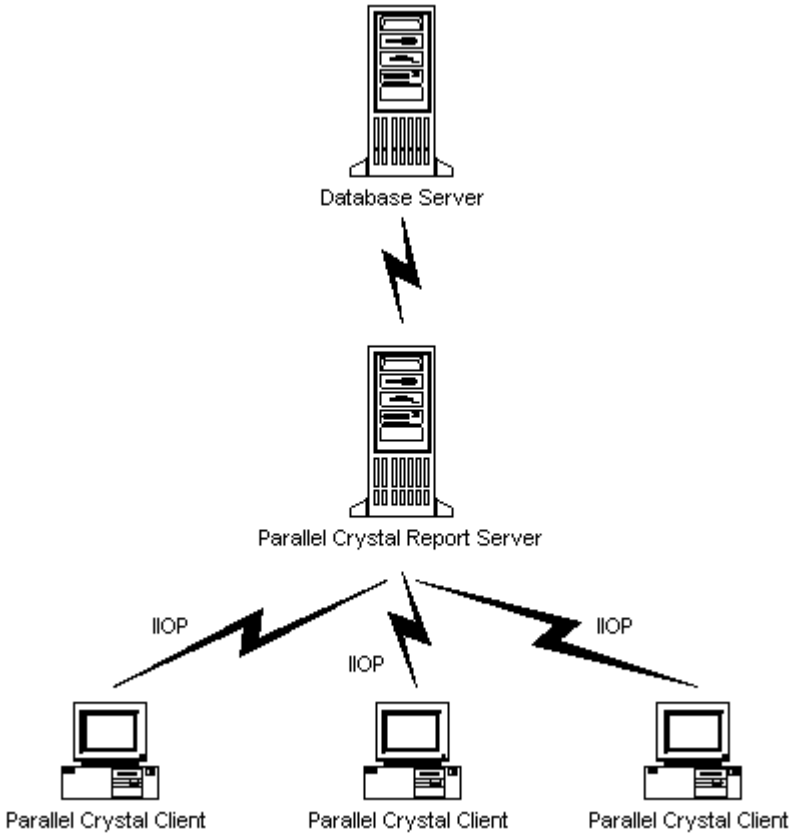


Figure 1.4

In this *three-tier* architecture, each client is connected to a Parallel Crystal Report Engine running in the Report Server. The report engine performs the customization commands for that client, but necessarily shares available processors and memory with other report engines running in the same server. The report files are also located within the report server, but the data records are retrieved from the database server.

The three-tier Parallel Crystal architecture provides may advantages, including

- The duplication of software within each client and the processing power required of each is reduced.
- All the reports are located within the report server making the reporting process is easier to administer.
- Both servers are dedicated to a single well-defined role and can be tuned for performance in that role.

Distributed Report Server Processing

Execution of the customization program is now distributed between the Parallel Crystal desktop clients and the Parallel Crystal Report Engine running in the Report Server. The process of "remoting" the customization call from the client to the server is accomplished with a technology called CORBA[1] that in turn uses a network protocol called IIOP[2] that is designed specifically for invoking remote API calls. Imagine for a moment, Example 1.1 on page 18, running in the client. When execution reaches the statement

```
PCREJob job = engine.OpenJob("Box.rpt");
```

the name of the report file is transmitted to the server and the corresponding Report Engine API call is made with the function `PEOpenPrintJob`. If you look up this function in the Seagate Crystal Reports Developer's Help information, you will find that it is declared as the C function:

```
short PEOpenPrintJob( char * reportFilePath )
```

When the function completes, it returns an integer that represents a "handle" to the print job, and this handle is returned to the PCRE client program. It may just be apparent that it is stored internally as an attribute (or property) of the Java `PCREJob` class.

Since Parallel Crystal uses CORBA/IIOP to transmit the arguments and results of API calls to and from the server, it follows that Parallel Crystal client program must be able to "talk CORBA". While this is true of the C, C++ and Java programming languages, it is not the case for important Windows development tools such as Visual Basic and Delphi. In these cases, Parallel Crystal clients are provided with an ActiveX Automation Server that provides a bridge between ActiveX and CORBA technologies. The Automation Server exposes the API functions through an object hierarchy that is oriented particularly to the requirements of object-based scripting languages such as Visual Basic. However, more conventional programming languages may use the same interface as long as they conform to the Microsoft Active X specification. Naturally, this includes Microsoft Visual C++ and Visual J++.

Parallel Crystal and Web Application Servers

Recent developments on the World Wide Web have led to the emergence of a new kind of server called a Web Application Server (WebAS). Web Application Servers aim to provide very large numbers of browser-based clients with facilities to access corporate data. Most of them incorporate rapid application development tools that allow the data to be extracted and presented in the form of dynamically composed web pages.

When Parallel Crystal is integrated with a WebAS, we get an "n-tier" application similar to that shown in Figure 1.5.

---

[1] CORBA stands for Common Object Request Broker Architecture.

[2] IIOP stands for Internet Inter-ORB Protocol.

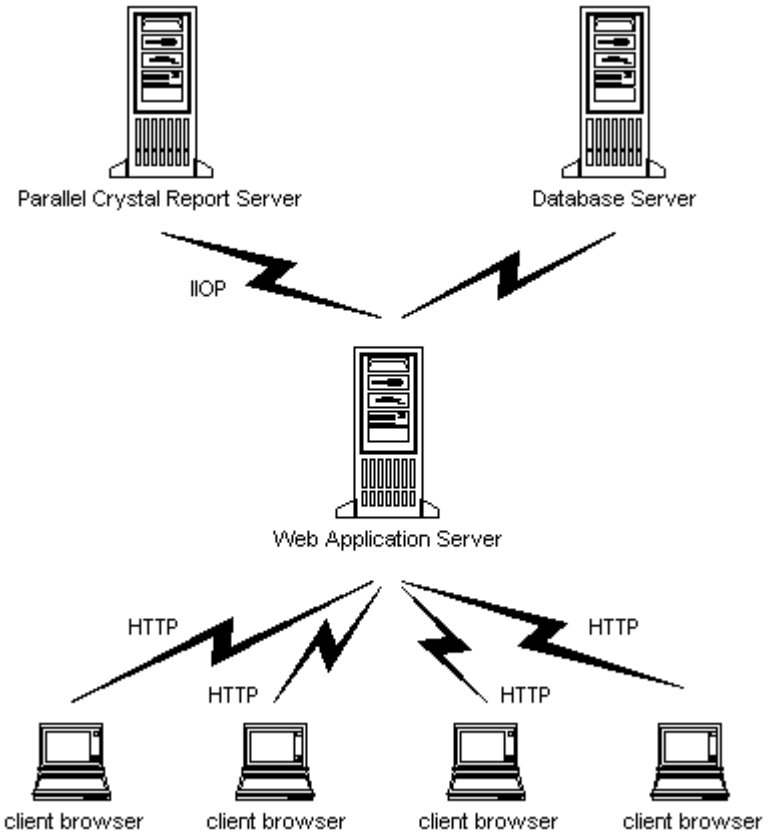


Figure 1.5

The core of the WebAS is a high performance, highly scalable web server that is capable of scheduling requests for web pages. In many cases however, the pages served are the result of dynamic composition performed by additional back-office tools invoked by the web server. These may include CGI Perl Scripts, server-side plugins, Java servlets or Microsoft Active Server Pages. The usual sequence of actions begins when the client submits a query using an HTTP request. The web server receives the request and dispatches it to the correct server-side utility. The utility analyzes the request, accesses a database and composes an HTML page to display the data, and returns the page to the web server. In the final step, the web server returns the page to the client.

When Parallel Crystal is integrated with a WebAS, it provides report customization, production and retrieval facilities that are accessible via your web browser. Because of the limited capabilities of the HTTP protocol, Parallel Crystal clients do not attempt to make customization API calls from the browser to the Report Server via the Web Server. Instead, all customization code is executed within the WebAS/Report Server framework and only the final report is returned to the browser.

Figure 1.6 illustrates the role of each component when Parallel Crystal is integrated with Microsoft's Active Server Pages technology.

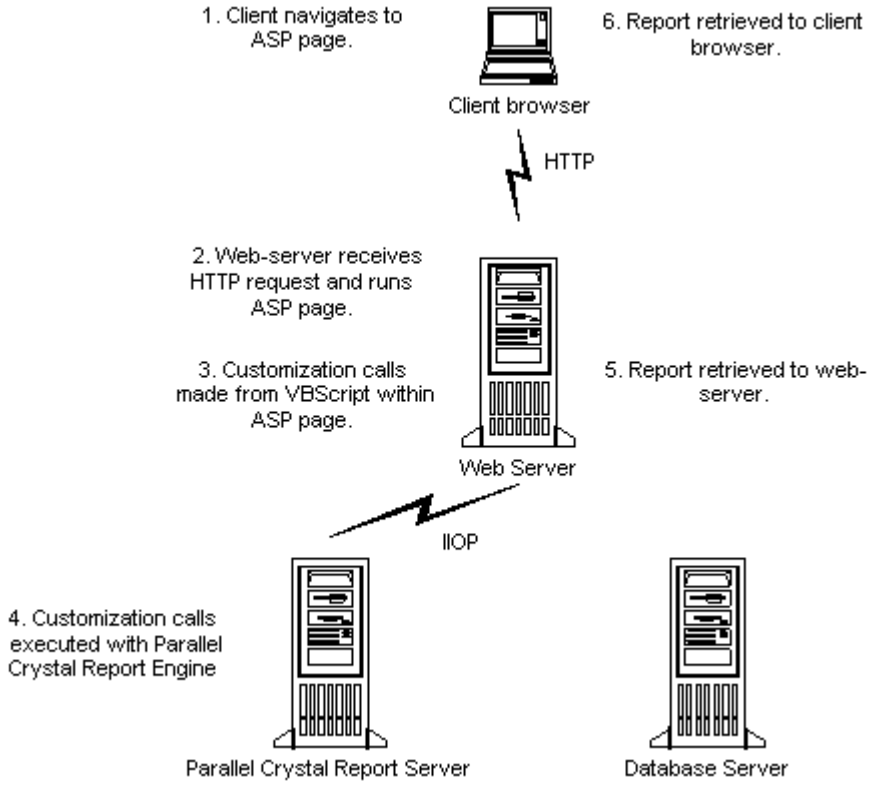

1. Client navigates to ASP page.
6. Report retrieved to client browser.
Client browser
HTTP
2. Web-server receives HTTP request and runs ASP page.
3. Customization calls made from VBScript within ASP page.
5. Report retrieved to web-server.
Web Server
IIOP
4. Customization calls executed with Parallel Crystal Report Engine
Parallel Crystal Report Server
Database Server

Figure 1.6

When the client browser navigates to an ASP page, the Microsoft IIS Web Server loads the page and runs the VBScript interpreter on the VBScript tags embedded in the page. The VBScript statements make customization calls to the Report Engine, and then the generated report is retrieved to Web Server. The report itself or its URL is then returned to the client's browser.

You should bear in mind that the Web Server, Report Server and Database Server need not all run on different machines. However, the configurations shown in Figures 1.5 and 1.6 allow maximum opportunity for performance tuning.

Parallel Crystal provides a networked environment for customizing and generating Crystal Reports. Clients are connected to Parallel Crystal Report Engines running in the Report Server and customization API calls are transmitted from client to server using CORBA/IIOP technology. A selection of client packages allows the Server to be accessed using a variety of development tools and software technologies.

Parallel Crystal Client and Server Components

Parallel Crystal contains a set of server components, and several distinct sets of client components. The installation additionally requires a web server, and may also require Adobe Acrobat tools for viewing PDF documents and the Sun JDK for Java development.

Note: The Parallel Crystal System Administrator's Manual is being developed to explain these requirements in detail.

The Parallel Crystal Server components include

- An implementation of CORBA/IIOP. Parallel Crystal uses Inprise Corporation's VisiBroker product that provides client/server connectivity through two components called the OSAgent and IIOP GateKeeper.

- The PCRE Report Engine Server. Each Parallel Crystal client is connected to a Parallel Crystal Report Engine that encapsulates the Print Engine DLL. The Report Engine is a CORBA server that exposes the customization interface to clients through the CORBA Interface Definition Language (IDL).

- The PCRE Gateway. The Gateway allocates Report Engines to clients, and can balance the requests from multiple clients across multiple Report Servers. The PCRE Gateway also has an interface that is expressed in CORBA IDL and is implemented as a CORBA server.

- The PCRE Service Control Manager. The Service Control Manager (SCM) allows Parallel Crystal to be run on the desktop or as an NT Service. The SCM also provides a number of intrinsic services to Parallel Crystal clients.

Normally, a Parallel Crystal installation will include a PCRE Service Manager, a Gateway, an OSAgent and an IIOP GateKeeper. In addition there will be a PCRE Report Engine Server for each connected client.

Parallel Crystal has distinct client packages for use with Java, Microsoft ActiveX and C++ development environments. The choice of which package to use depends upon your preferences for development languages and technologies.

The principal features of each Parallel Crystal client include

- The Parallel Crystal Java Client is provided as a single Java package called `com.mobileapps.pcre`. It contains an object-oriented interface to the customization API and a collection of supporting classes that provide server location and connection management, and error reporting and recovery. The package may be used within Java applications or applets.

- The Parallel Crystal C/C++ Client is a simple class library that offers dual interfaces for C++ and languages supporting C-style function linkage. It offers the original C interface to the customization API and an object-oriented interface that is similar to the Java package. It also includes additional classes for connection management, error reporting and recovery.

- The Parallel Crystal ActiveX Client provides a COM compliant interface to Microsoft development tools such as Visual Basic, Visual Basic Scripting Edition, Visual J++ and Visual C++. This is achieved through the PCRE Automation Server that presents the customization API as a hierarchy of COM automation objects.

These client packages are arranged in a layered architecture that allows for upwards extension. The arrangement of these layers is shown in Figure 1.7.

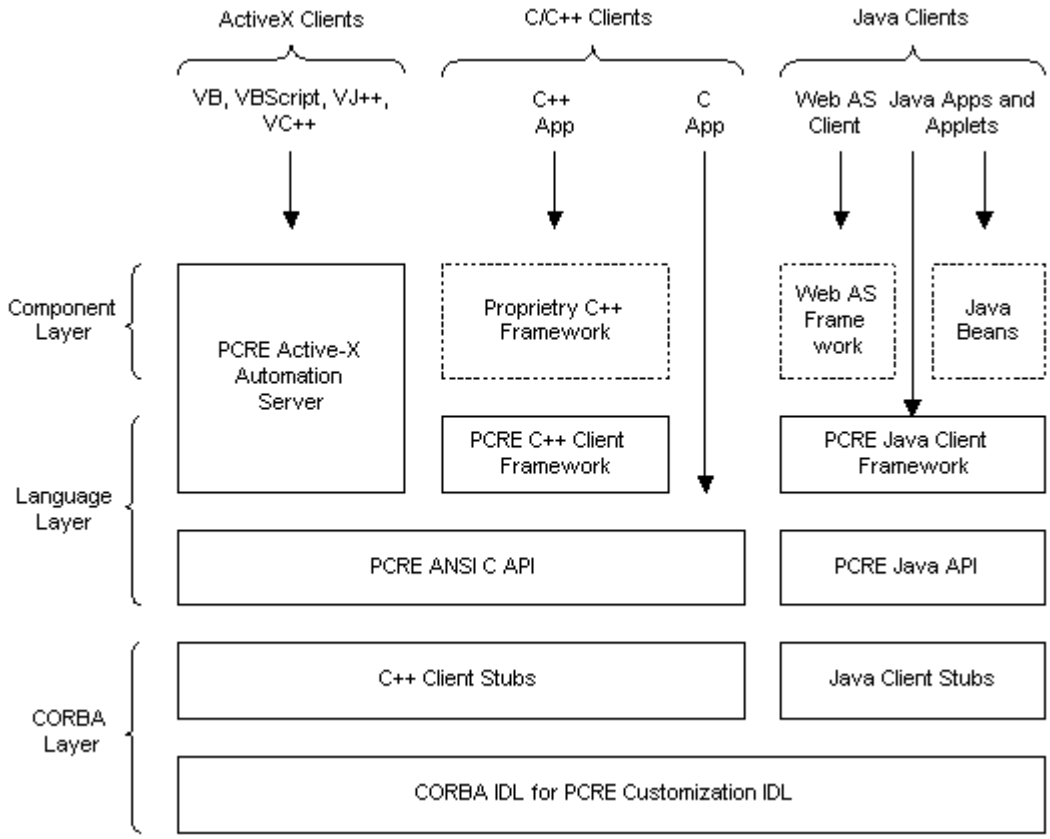


Figure 1.7

Corba Layer

The CORBA Layer consists of a common CORBA IDL interface for the PCRE Customization API that is shared by all clients. Internally, the interface is incorporated into each client as a set of "stub-functions" which marshal the arguments for the various API calls and send them across the network to the Report Server. The stubs also retrieve results and handle errors when they occur. Neither the stubs nor the IDL are normally exposed to users. That means that you do not have to be a CORBA programmer to use Parallel Crystal.

Language Layer

The Language Layer exposes the PCRE customization API in a language-dependent manner. For C++ and Java applications the language layer provides a moderately object-oriented version of the API through classes that implement "print engine" and "print job" objects. Additional classes allow client applications to connect to servers, and provide exception-oriented error handling which is natural for these languages. By contrast, ANSI C applications interface to a traditional C "header file" which contains the declarations of all the API functions. Error handling is provided using a system of returned error messages and codes.

Component Layer

The Component Layer exposes the PCRE customization API using industry standard component object models. Currently, the language layer includes the PCRE ActiveX Automation Server that supports the Microsoft COM standard. The Automation Server re-expresses the customization API as a hierarchy of objects offering "dual" interfaces that can be used by Microsoft scripting languages such as VBScript and JScript, and Microsoft compiled clients such as Visual C++ and Visual J++. Error handling is provided using the standard COM interfaces for providing error information. Future versions of Parallel Crystal may add support for Java Beans and additional WebAS-specific frameworks to the component layer.

Some final points about layered architecture and various client frameworks

- The client frameworks provided span Microsoft and non-Microsoft technologies. For example you can choose whether to use Java via Microsoft Visual J++, or whether to use a "100% pure" Java solution such as the Sun JDK.

- The C++ client framework has a small footprint, and does not introduce dependencies upon any particular C++ development environment.

- Neither the C++ nor the Java client frameworks make assumptions about the end user application. They can be used in graphical and non-graphical applications.

- All client frameworks are thread-safe. In particular, the ActiveX Automation Server conforms to the Apartment Level Threading Model required by Microsoft Active Server Pages, and the C++ and Java clients support the construction of multi-threaded applications.

Parallel Crystal PDF Library

Parallel Crystal provides users with a form of output known as PDF (Portable Document Format). This output is useful when exacting detail is required for reports, because the report looks the same and prints the same from any type of computer.

The Parallel Crystal PDF library is the primary method for outputting a report to PDF format. This library provides several distinct advantages for a server environment.

The advantages include

- PDF library is completely scalable.
- PDF library is faster than Acrobat Distiller (with the exception of charts).
- The library maintains accurate page placement of objects.
- The library will soon support linearized PDF output.

PDF Output via Adobe Acrobat Distiller

MAS also supports PDF generation via Adobe's Acrobat Distiller. Support for Distiller is being phased out. Distiller has several issues that have prompted the migration to the MAS PDF library.

The Distiller issues include

- Installation Issues - If you do not have a PostScript printer driver installed and set to default, then Distiller will not install on your computer.
- Stability issues - Distiller is not designed to operate in a server environment where you need to run as a service, have multiple instances of Distiller running on separate jobs, or multiple Distiller's running on a single job to speed output.
- Formatting issues - The page placement of objects is sometimes inaccurate.
- Thread Safety - Distiller is a single-threaded application, which means that no matter how many Distillers are started, they serialize the output. This slows everything down
- Size Limitations - There seems to be a size limit to the postscript file that can be created by Crystal. This is important because postscript is the input to Distiller. Users cannot create a postscript file that has more than about 7500 pages in it; so long reports using Distiller are not possible.

Chapter 2
Getting Started with Java Clients

In this chapter we describe how to customize and generate reports using Java. You may use any "pure" Java environment with Parallel Crystal, so long as it incorporates a JDK 1.2.x Java Virtual Machine. You may also use Microsoft's Visual J++ environment, so long as you don't intend using the ActiveX/COM compiler extensions[1].

The chapter is divided into two major sections

- In the first section entitled Using Parallel Crystal in a Java Application we describe how to build a Parallel Crystal client that runs as a stand-alone Java application program. The section contains example code fragments that show you how to connect to the Parallel Crystal Report Server and how to make simple customization calls to generate reports formatted as PostScript, PDF and HTML documents.

- In the second section entitled Using Parallel Crystal in a Java Applet we describe how to build a Parallel Crystal client that runs as an applet in a Java-enabled web browser. You can use any web browser so long as it incorporates a Java 1.1.x Java Virtual Machine.

Both application and applet clients are constructed using a Java package called `com.mobileapps.pcre`. The next chapter entitled Using the Java Client Package describes the Java client interface in detail.

In both sections, we use a sample Crystal report file called `Box.rpt` that is distributed with Parallel Crystal in the `SampleReports` directory of the product installation. The file already incorporates its own data so the report may be produced directly without the need to connect to a database.

[1] We describe how to use Visual J++ ActiveX/COM extensions with Parallel Crystal in Chapter 5.

Using Parallel Crystal in a Java Application

In this section we describe how to construct a Java application program that generates a report as a PostScript, PDF or HTML document. For simplicity, we assume that the Java program has a simple character-based user interface. Once you've mastered the material in this chapter, you can refer to the section entitled Print Job Outputs in Chapter 3 to learn how to generate reports in other formats.

Before looking at some example code we need to clarify details of a typical Parallel Crystal Report Server installation.

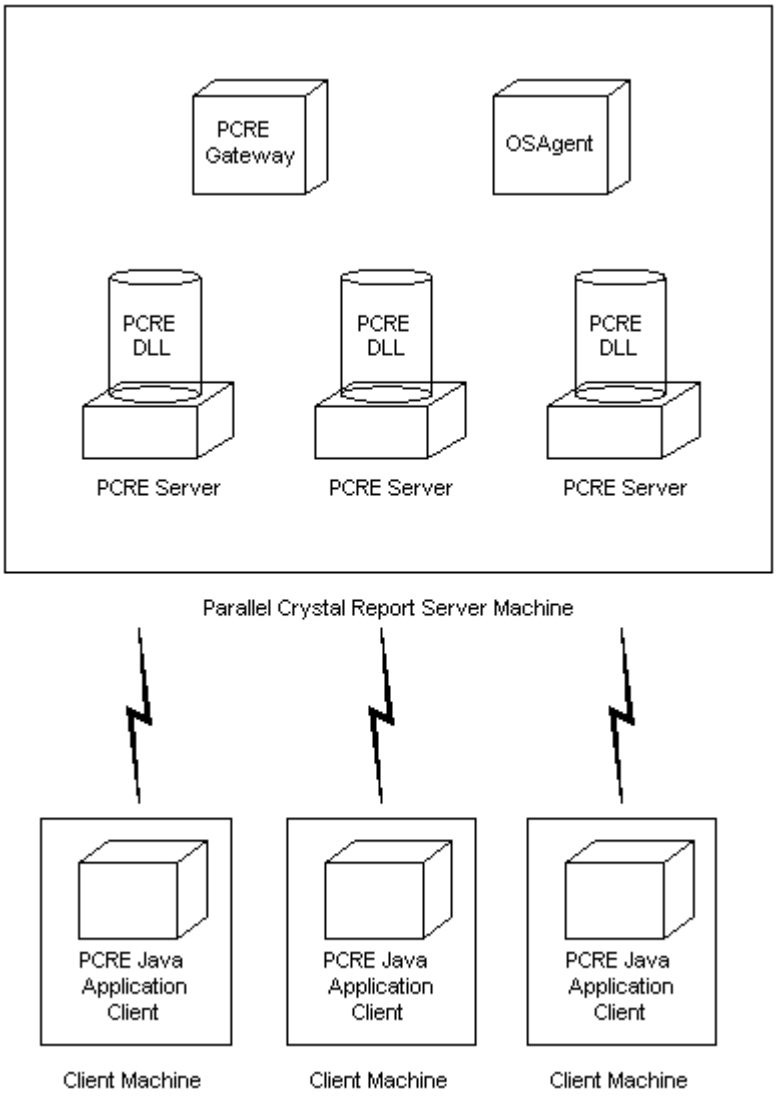


Figure 2.1

Figure 2.1 on page 30, illustrates the following important points

- The Parallel Crystal Report Server is normally hosted within a single powerful multi-processor server machine running Windows NT.

- A Parallel Crystal Java Application Client normally runs in a less powerful client machine such as a desktop PC running Windows 95.

- Each Java Application Client is connected to a corresponding PCRE Report Engine in the Report Server machine. Each Report Engine includes a copy of the Parallel Crystal Report Engine (PCRE) DLL. An agent on the Report Server called the PCRE Gateway establishes Client/server connections.

- The Java Application Client performs report customization and generation by making calls to the PCRE DLL. These calls are transmitted from the client to the server, and then the results are transmitted back to the client.

- Because report customization is performed within the Report Server, the Crystal report files must be present on that machine, or be available through networked file store.

- Reports are produced on the Report Server machine. However, Parallel Crystal provides a number of mechanisms for viewing and then retrieving reports back to the client machine.

- When a Java Application Client is started, it is allocated a Report Engine when it connects to the Report Server. The Report Engine is terminated when the client disconnects from the Server.

- Multiple Report Engines execute "in parallel" on the Report Server. However, the extent to which truly concurrent behavior is sustained depends upon a combination of factors including the number of available processors, the number of connected clients, the overall loading on the machine, and the resulting contention for shared resources.

We're now in a position to look at a simple Java Application Client that generates a report as a PostScript document.

Generating a Simple PostScript Report

The sample Java program shown in Example 2.1 generates a report from the `Box.rpt` report file and sends it to a PostScript printer. We'll describe it line by line.

```
import com.mobileapps.pcre.*;                                    //1.

class PostScriptExample
{
   public static void main( String [] args )
   {
      PCREApplicationClient client = null;
      try
      {
      client = new PCREApplicationClient("example.host.com");
                                                                 //2.

        client.Connect();                                        //3.
        PCREEngine engine = client.OpenEngine();                 //4.
        PCREJob job = engine.OpenJob(
           "C:\\MobileApps\\PCRE\\SampleReports\\Box.rpt");
                                                                 //5.

        job.OutputToPrinter(1);                                  //6.
        job.Start();                                             //7.
        job.Close();                                             //8.

        System.out.println("Report generated");                  //9.
        engine.Close();                                          //10.
       }
       catch ( PCREError ex )                                    //11.
       {
         System.out.println(“Exception: +ex);                    //12.
       }
       finally
       {
         if (client != null) client.Quit();                      //13.
       }
   }
}
```

Example 2.1

Each numbered paragraph below corresponds to the same numbered code statement

1. Parallel Crystal Java clients use the Parallel Crystal Client package called `com.mobileapps.pcre`. In order to access the public classes in this package, you must import it into your Java program by means of the import statement shown. You must also ensure that the form of the import statement is consistent with the setting of the CLASSPATH in your Java development environment. The PCRE package is currently distributed as a jar file whose full path name (assuming installation on drive C) is:

   ```
   C:\MobileApps\PCRE\jars\PCREjavaclient.jar
   ```

   Your CLASSPATH must include this path name to access the package. The correct adjustments to your CLASSPATH are normally made during installation.

2. The declaration creates an instance of the PCREApplicationClient class. The constructor call specifies the host name of the machine running the Parallel Crystal Report Server as an argument string. This constructor assumes that the gateway is broadcasting its initial reference to the default port.

   Note: The value `"example.host.com"` is only an example, and must be replaced with a valid host name.

3. This statement connects the client to the Report Server running on the machine specified in Step 2. It is included in a try-catch block so that if the call to `Connect` fails, an exception will be raised and control will transfer to the exception handler at statement 11. We'll discuss connection failures in greater detail in the next chapter, but the most common reasons are that the Parallel Crystal Report Server is not running on the machine identified by the host name, or you specified an incorrect host name.

4. This statement declares a variable that represents a PCRE print engine. The variable is initialized with the value returned by the call to the `OpenEngine` method of the `client`. In statements 2 through 4 you can see a progression: get a client, connect the client to the Report Server, then open a print engine on the Report Server.

5. The Parallel Crystal Report Engine allows reports to be customized and started by running print jobs. Each job is opened initially by specifying the name of the report file that is to be used to generate the report document. In the example, we specify the file using a full path name for a typical Parallel Crystal installation.

6. Once a print job is opened, it may be customized with calls to the methods of the `PCREJob` class. This statement calls the `OutputToPrinter` method to send the output to a PostScript printer. The argument value specifies the number of copies to be printed.

7. The call to the `Start` method generates the report document. When the call returns, the report document will be complete, but printing will not necessarily have finished.

8. When you are finished with a print job, you should call the `Close` method to release resources held in the Report Server.

9. This statement is executed if all previous statements executed without error. If an error occurs in an API call, the corresponding method of the `PCREEngine` or `PCREJob` class will throw an exception and control will transfer to the exception handler at statement 11.

10. When we are done with all print jobs, the print engine is closed with a call to its `Close` method.

11. The exception handler is executed if an expception occurs in any of the preceding method calls. We will discuss exception handling in detail in Chapter 3, but for the present, its sufficient to note that by specifying the base class `PCREError` in the handler, we can trap all exception classes raised by the Java Client.

12. In this simple example, we choose to handle the exception by displaying the exception on the screen.

13. The last statement calls the `Quit` method and disconnects the client from the Report Server. You should call this method whenever you are finished with generating reports, since it will automatically terminate the Report Engine and free up resources in the Server. Here, it is included in a finally clause so that it is sure to be called before the program completes.

This example may be summarized as a set of rules that you can use to develop your own applications

- Access the Parallel Crystal Java Client package by importing `com.mobileapps.pcre.*`

- Create an instance of the `PCREApplicationClient` class using the host name (or IP address) of the machine hosting the Report Server.

- Call the `Connect` method to connect your client to a Report Engine running in the Server.

- Open a print engine by calling the `OpenEngine` method of your client.

- Use the `PCREEngine` instance returned by `OpenEngine` to open one or more print jobs. Each job is specified by supplying the full path name of the report file on the Report Server.

- Perform customization and report generation calls for each print job using the methods of the `PCREJob` class.

- Close all your print jobs when they have completed using the `Close` method for each job.

- Close your print engine.

- Call the `Quit` method to terminate your Report Engine and disconnect from the Report Server.

Trouble Shooting the Simple Client

This section details some of the most common problems in programming a simple client. Refer to the sample code from Example 2.1 on page 32.

Incorrect Host Name Declaration

Suppose you supplied an incorrect host name in the declaration of the `PCREApplication` client variable.

For example, suppose you typed:

```
client = new PCREApplicationClient("bad-host name");
```

This statement will be executed correctly, but the subsequent call to `Connect` on line 3 will throw an exception and control will transfer to the handler at line 12.

This statement will then print the following line:

```
PCRE Gateway error: Connect: cannot connect client to Gateway
    Service on host "bad-host name".
```

The error message indicates that an attempt to connect the client to the Gateway on the nominated host failed.

This error message usually indicates one of a number of things, including:

- The Report Server host name you supplied in the `PCREApplication` client constructor call was invalid.
- The host name was valid, but Parallel Crystal was not running on the Report Server when the connection was attempted.
- The Gateway wasn't broadcasting its initial reference on the default or provided port.
- There was some other communication problem.

Invalid Report File Name

The second most common error in programming a simple client is to specify an invalid name for the report file.

For example suppose that instead of line 5 you typed:

```
PCREJob job = engine.OpenJob
    ("C:\\MobileApps\\PCRE\\SampleReports\\junk.rpt");
```

When this statement is executed, the report pathname will be supplied to the Print Engine DLL that will subsequently fail because the report file does not exist. In this case, the DLL returns an exception to the client that causes control to transfer to the handler at line 12.

This statement prints the following line to the standard output stream:

```
PCRE API error: invalid file name.
```

The "PCRE API error" prefix indicates that the Print Engine DLL raised the error. Generally this is an indication that something went wrong in a Print Engine API call, and the most likely cause is an invalid argument value. Unfortunately, the error messages returned by the Print Engine DLL are quite terse. One way to compensate for this is to enclose smaller groups of calls within separate try-catch blocks so that you can be sure which call failed.

Missing Report Destination Specification

A third error that is easy to make is to forget to specify the destination for your report. In other words, once it's generated, where do you send it? In the simple client, line 6 specified that the final report was to be sent directly to a PostScript printer rather than be stored in a file. If this line is omitted, then the Print Engine DLL will raise an exception when the client executes the `Start` command on line 7. Control will transfer to the handler on line 12 and the following line will be printed to the standard output stream:

```
PCRE API error: no print destination specified.
```

This message indicates that the Print Engine DLL could not figure out what to do with your report. You must respond by specifying output to a printer or to a disk file. We'll explore the most common solutions in the next two sections.

Overall, the simple client illustrates the importance of exception handling in Parallel Crystal client programming. In fact because of the nature of the Java programming language, you will not be able to construct client programs without including try-catch blocks around most API calls. We'll describe the error handling facilities in the PCRE package in more detail in the next chapter.

Generating a Simple PDF Report

Adobe's Portable Document Format (PDF) is widely used as an alternative to PostScript or HTML on the Internet. It provides PostScript quality reproduction (which is far superior to HTML), using a more compact document encoding format.

Example 2.2 on page 37, illustrates a simple Java client that outputs a version of the Boxoffice report to PDF. It is in fact identical to Example 2.1 with the exception of the single line highlighted.

```
import com.mobileapps.pcre.*;                                    // 1.

class PDFExample
{
  public static void main( String [] args )
  {
    PCREApplicationClient client = null;
    try
    {
      client = new PCREApplicationClient("example.host.com"); //
2.
      client.Connect();                                          // 3.
      PCREEngine engine = client.OpenEngine();                   // 4.
      PCREJob job = engine.OpenJob(
         "C:\\MobileApps\\PCRE\\SampleReports\\Box.rpt");        // 5.

       job.OutputToPDF(
          "C:\\MobileApps\\PCRE\\SampleOutputs\\Box.pdf",
            null);                                               // 6.
       job.Start();                                              // 7.
       job.Close();                                              // 8.

       System.out.println("Report generated");                   // 9.
       engine.Close();                                           // 10.
    }
    catch ( PCREError ex )                                       // 11.
    {
      System.out.println("Exception: "+ex);
    }
    finally                                                      // 12.
    {
      if (client != null) client.Quit();                         // 13.
    }
  }
}
```

Example 2.2

A PDF report is generated by calling the `OutputToPDF` method of the `PCREJob` class and supplying the name of the output file as an argument. The PDF document is produced when the job is started at line 7, and is stored in the file `C:\MobileApps\PCRE\SampleAppsOutput\Box.pdf`. In order to display the file, you must have access to a PDF viewer such as Adobe's Acrobat Reader. Most Internet browsers have this viewer pre-installed as plugin.

The `OutputToPDF` method call uses the Dynalivery PDF Library that extends the Parallel Crystal Report Engine with facilities to render reports directly in PDF. It is possible to provide a set of PDF output options as a second argument to `OutputToPDF`. However, these options require detailed knowledge of PDF and further discussion is deferred to the Chapter 3. In practice, the default operating options are sufficient for most purposes and are automatically selected when the second argument to `OutputToPDF` is set to `null`.

Generating a Simple HTML Report

If the reports you wish to generate are web-based and intended for display exclusively through web browsers, then it may be appropriate to generate the report as an HTML file.

Example 2.3 below illustrates how the Box Office report can be generated in HTML. Once again, the code differs only slightly from the preceding examples.

Generating a report in HTML looks a little more complicated since two extra statements are required rather than one. In the example line 6 creates an instance of the `PCREHTMLJobExportInfo` class using the name of the HTML file as an argument to the constructor. This object is passed to the `ExportTo` method that informs the Print Engine DLL that the output for the print job is to be "exported" to HTML. The HTML file is generated when the job is started at line 7. Notice that when a graphical report is generated in HTML, any embedded images will be stored in additional JPEG files in the same directory. If the name of the HTML file is `Box.html`, then the JPEG files are named `Box001.jpeg`, `Box002.jpeg`, and so on.

The Parallel Crystal Report Engine supports a large number of additional "export formats" which allow reports to be formatted for a variety of front-office tools. We'll describe these formats in greater detail in Chapter 3.

```
import com.mobileapps.pcre.*;                                    // 1.

class HTMLExample
{
  public static void main( String [] args )
  {
    PCREApplicationClient client = null;
    try
    {
      client = new PCREApplicationClient("example.host.com");
                                                                 // 2.

      client.Connect();                                          // 3.
      PCREEngine engine = client.OpenEngine();                   // 4.
      PCREJob job = engine.OpenJob(
         "C:\\MobileApps\\PCRE\\SampleReports\\Box.rpt");        // 5.

      PCREHTMLJobExportInfo info = new PCREHTMLJobExportInfo(
         "C:\\MobileApps\\PCRE\\SampleOutputs\\Box.html");       // 6.
      job.ExportTo(info);                                        // 6a.
      job.Start();                                               // 7.
      job.Close();                                               // 8.

      System.out.println("Report generated");                    // 9.
      engine.Close();

                                                                 // 10.
    }
    catch ( PCREError ex )                                       // 11.
    {
      System.out.println(“Exception: “+ex);                      // 12.
    }
    finally
    {
    if (client != null) client.Quit();                           // 13.
    }
  }
}
```

Example 2.3

Summary

- The preceding four sections have described how to construct simple Java clients that connect to a Parallel Crystal Report Server running on a named host. An exception will be generated if you give an incorrect host name, or Parallel Crystal is not in fact running on that machine.

- Reports are generated by getting access to a print engine, and then opening a print job. When the print job is opened, you supply the full path name of the report file on the Report Server.

- You must tell the Print Engine what to do with the report once it is generated.

- If you want to print a PostScript report, call the `OutputToPrinter` method of the `PCREJob` class.

- If you want to generate a PDF report, call the `OutputToPDF` method and specify the full path name of the PDF file.

- If you want to generate an HTML report, create an instance of a `PCREHTMLJobExportInfo` class and supply the full path name of the HTML file to the constructor. Then call the `ExportTo` method and pass the export-info object as an argument.

- Report output files are not generated until the `Start` method is called.

- You must enclose most PCRE method calls within try-catch blocks in order to catch and handle exceptions raised by the PCRE Java package and the Print Engine DLL.

Using Parallel Crystal in a Java Applet

In this section we describe how to write Java applets that connect to a Parallel Crystal Report Server, and how to retrieve the generated report back to your browser. You are probably familiar with the applet concept, but we will clarify the rules for applet programming and then show how they are extended to Parallel Crystal clients.

In programming terms, a Java applet is a class that extends a user interface Panel called `java.applet.Applet`. This class has four methods called `init`, `start`, `stop`, and `destroy`, that are called as you navigate from page to page. When an HTML page containing an <applet> tag is loaded into your browser, the class file for the applet is loaded into the browser's Java virtual machine and the `init` method is called. If you subsequently browse to another document using the left or right arrow keys or by entering a new URL, then the browser calls the `stop` method before leaving the page containing your applet. Conversely, if you return to the page containing your applet, the `start` method is called again. Finally, if you terminate your browser, then the `destroy` method is called. You can control the behavior of your applet by overriding the `init`, `start`, `stop` and `destroy` methods with your own implementations.

Although there are several possible approaches, we recommend that you write a Parallel Crystal client applet using the framework shown in Example 2.4 on page 42.

```
import java.awt.*;                                          // 1.
import java.applet.*;                                       // 2.
import com.mobileapps.pcre.*;                               // 3.

public class PCREApplet extends Applet                      // 4.
                        implements PCREConsole {            // 5.

   private PCREAppletClient client = null;                  // 6.

   // PCREConsole implementation methods ...                // 7.

   // Applet implementation methods ...
   private void RunJob()                                    // 8.
   { customize and run report }

   /* Override called when applet loaded. */
   public void init()                                       // 9.
   { create applet user interface }

   /* Override called when page is visited. */
   public void start()                                      // 10.
   { connect client to the PCRE report server }

   /* Override called when page is departed. */
   public void stop()                                       // 11.
   { disconnect client from the PCRE report server }

   /* Override called when browser is terminated. */
   public void destroy()                                    // 12.
   { disconnect client from the PCRE report server }

   /* Despatch handler. */
   public boolean action( Event evt, Object arg )           // 13.
   {
      // despatch events ...
      return true;
   }
}
```

Example 2.4

Each numbered paragraph below corresponds to the same numbered code statement

1. Java applets differ fundamentally from the Java applications because they are effectively part of the user interface provided by your browser. This means that your applet must use one of the Java User Interface toolkits in order to present its output. Simple interfaces can be constructed using the Java Abstract Windowing Toolkit (AWT), but for commercial quality interfaces your are more likely to require the new Java Foundation Classes.

2. You must build your applet by extending the `Applet` class from `java.applet`. Consequently, you must import this package.

3. You must import the `com.mobileapps.pcre` package to access the Parallel Crystal Java client classes.

4. The `PCREApplet` class must be declared as a sub-class of `java.applet.Applet`. This provides access to the `start()`, `stop()`, `init()`, and `destroy()` methods and allows you to override them.

5. The `PCREApplet` class must implement the `PCREConsole` interface. We'll explain this in more detail below, but basically it allows the error reporting facilities within the PCRE package to be adapted to your user interface.

6. PCRE applet clients must declare and create an instance of the `PCREAppletClient` class. This class adapts the PCRE client to the rather restricted "Java Sandbox" in which applets are forced to run.

7. You must provide an implementation of the `PCREConsole` interface. This interface declares the following seven abstract methods:

```
public abstract void PCREOutputMsg( String msg );
public abstract void PCREFatalErrorMsg( PCREError error );
public abstract void PCREInternalErrorMsg( PCREError error );
public abstract void PCREServerErrorMsg( PCREError error );
public abstract void PCREAPIErrorMsg( PCREError error );
public abstract void PCREGatewayErrorMsg( PCREError error );
public abstract void PCREClientErrorMsg( PCREError error );
```

   These methods give you an opportunity to adapt the error reporting requirements of the PCRE package to your user interface. For example, you may wish to log client, Server, API and Gateway error messages in a display panel and then continue processing. However, fatal and internal error messages should probably be treated differently. The `PCREError` class is derived from `java.lang.Exception` and has property methods called `getCode()`, `getText()`, and `getKind()` that return details of the exception. If you are not too concerned with the quality of error handling in your applet, you can treat all errors in the same way and map the implementation of these methods to a common handler.

8. The applet implementation methods contain the core of your applet processing. In the example, this is represented by a single method called `RunJob()` that we'll expand below.

9. The purpose of the overriding `init()` method is to establish the user interface provided by the applet. This may be a simple text panel to display messages, or something much more complicated. Whatever you decide, you need to have at least elementary knowledge of Java user interface programming to provide the implementation of this method. Remember `init()` is called when the applet is first loaded into the browser. So it seems appropriate to build and display the user interface at that point.

10. The `start()` method is called each time the browser displays the page containing the applet. This means it is called immediately after `init()` and each time you subsequently return to the page. Within the framework illustrated, the purpose of `start()` is to create a PCRE applet client and connect it to a Report Server. The connection will exist for as long as the applet is displayed.

11. The purpose of the `stop()` method is to break the connection established by `start()`. This means that whenever you navigate away from the page, the Report Engine allocated to the applet client is terminated and the applet client is destroyed. When you navigate back to the page the browser will invoke `start()` again and a new applet client will be connected to a new Report Engine on the Server.

12. The `destroy()` method is called if you terminate your browser while displaying the page containing the applet. It should therefore terminate the Report Engine and destroy the existing applet client.

13. Between calls to `start()` and `stop()` the applet client must be accessed via the user interface. For example, if we have a button labeled "Run", then we could invoke the `RunJob()` method through an action handler as follows:

```
public boolean action( Event evt, Object arg )
{
   if ( "Run".equals(arg) ) {
      RunJob();
      return true;
   } else if ( "Quit".equals(arg) ) {
      QuitClient();
      return true;
   }
}
```

    The underlying Java Abstract Windowing Toolkit will invoke this method whenever buttons labeled "Run" and "Quit", that are part of the user interface contained within the applet Panel, are pressed. By trapping and dispatching events as shown we can invoke the methods within the applet that perform the report customization and generation.

This framework provides a transient connection to a Report Server. We'll provide a little more implementation detail and then describe how it might be adapted to support a persistent server connection. Let's look first at a typical implementation of `start() that` creates an applet client and connects it to the server.

```
private method ConnectClient()
{
   try {
      client = new PCREAppletClient(this, this);
      client.Connect();
      UIDisplayText("Connect to report server");
   } catch ( PCREError error ) {
      error.Report();
      client = null;
   }
}
public void start() {
   ConnectClient();
}
```

Note that `start()` calls `ConnectClient()` that clearly does the real work. On return to `start(), client` is a reference to an instance of a connected `PCREAppletClient,` or is `null` if the call to the `Connect()` method failed.

The client is created by a call to the `PCREAppletClient` constructor that has the following signature:

```
public PCREAppletClient( PCREConsole console,
                         java.applet.Applet applet );
```

Since the `PCREApplet` class both implements the `PCREConsole` interface and extends the `java.applet.Applet` base class, a reference to the current instance is supplied by passing the current class instance `this` for both arguments. The first argument provides a reference to your implementation of the `PCREConsole,` and the second argument provides a reference to your applet. The underlying client/server CORBA substrate uses the applet reference to manage network connections in a manner that is consistent with restrictions imposed by the applet's Security Manager.

If the `Connect()` method returns correctly, then the applet client will be connected to a Report Engine which is running on the same machine as the web server that served the applet to the browser.

So if you originally browsed to:

`http://www.example.host.com/applets/pcreapplet.html`

then your Report Engine will be running on the host "mantis.acm.com".

Having created an applet client and connected it to a Report Engine, we can now use the Engine to produce a report. Typically we have to do this in response to input from the user interface, so an implementation of the `RunJob()` method that is called when the “Run” button is pressed, is shown below.

If you compare the code in `RunJob()` to that in Example 2.3 on page 39, then you'll see that the sequence of calls to produce the HTML formatted report is identical. In other words, the difference between PCRE applet client programming and PCRE application client programming is the difference between Java applets and applications. The PCRE API calls that you make to customize and generate reports remain the same.

```
private void RunJob()
{
   // Check the client is connected.
   if ( client == null ) {
      UIDisplayText("Client not connected");
      return;
   }
   // Set success/fail indicator.
   boolean ok = false;

   try {
      PCREEngine engine = client.OpenEngine();
      PCREJob job = engine.OpenJob(
         "C:\\MobileApps\\PCRE\\SampleReports\\Box.rpt");

      PCREHTMLJobExportInfo info = new PCREHTMLJobExportInfo(
         "C:\\MobileApps\\PCRE\\SampleOutputs\\Box.html");
      job.ExportTo(info);
      job.Start();
      job.Close();

      UIDisplayText("Report generated");

      engine.Close();
      ok = true;
   } catch ( PCREError error ) {
      error.Report();
   }

   // Display the generated report.
   if ( ok )
      ShowReport("C:\\MobileApps\\PCRE\\SampleOutputs\\Box.html");
}
```

Having generated an HTML report, it seems natural to retrieve it and display it in your browser. The method `ShowReport` does this by generating a retrieval URL using the getDocumentBase method to generate the host and protocol components.

```
private void ShowReport( String report )
{
   try {
      URL docURL = getDocumentBase();
      URL rptURL =
         new URL(docURL.getProtocol(), docURL.getHost(),
                 docURL.getPort(), "/" + report);
      UIDisplayText("Retrieving " + rptURL);
      getAppletContext().showDocument(rptURL, "_blank");
   } catch ( Exception e ) {}
}
```

Notice the extra "/" character which has to be inserted in the final URL to separate the port component from the file name component. If you forget this, your browser will display an error dialog when you call `showDocument`.

Once the report has been generated and retrieved, it may be time to navigate to another URL or quit the browser altogether. In either case, we want to terminate the Report Engine to release resources on the Server, and to destroy the applet client. This is achieved with the following simple `QuitClient` method that is called from the applet's, `stop` and `destroy` override methods.

```
private void QuitClient()
{
   if ( client != null ) {
      client.Quit();
      client = null;
      UIDisplayText("Client disconnected from report server");
   }
}

private void stop()
{
   QuitClient();
}

private void destroy()
{
   QuitClient();
}
```

With this framework, the client-server connection will be severed whenever you navigate away from the page containing the applet. If you return, a new client will be created by the call to `start` and it will be connected to a new Report Engine. Notice that by this time you have lost the references to the `PCREEngine` and `PCREJob` class instances that were opened within the first client, since they were local variables of the `RunJob` method. If you had stored them as instance variables in the `PCREApplet` class, they would now reference class instances from a "severed connection" and an attempt to invoke their methods would cause exceptions to be thrown from the PCRE package.

It is possible to generate and retrieve PDF reports by changing the code of `RunJob` to call `OutputToPDF` instead of passing a `PCREHTMLExportInfo` reference to `ExportTo`:

```
job.OutputToPDF("C:\\MobileApps\\PCRE\\SampleOutputs\\Box.pdf");
job.Start();
job.Close();
```

This code is identical to that in Example 2.3 on page 39. The `ShowReport` method will also work but you need to be sure that your web server can serve PDF files, and that your browser is equipped with an Acrobat Reader plugin to display the file.

<Applet> Tag Details

In order to use the PCRE package from an applet, your HTML <applet> tag must specify certain mandatory attributes. For example, the applet in Example 2.4 on page 42 should be accessed with the following sequence of HTML tags.

```
<applet
   name=PCREApplet code=PCREApplet id=PCREApplet
      width=600 height=400 >
   <param name=title
      value="PCRE Applet Example" >
   <param name=org.omg.CORBA.ORBClass
      value=com.visigenic.vbroker.orb.ORB>
   <param name=ORBalwaysProxy value=true>
</applet>
```

The initial <applet> tag must include the `code` attribute that identifies the applet's class file relative to the web server's root directory. For example, if the web server root is `C:\InetPub\wwwroot` then the applet code will be retrieved from the file `C:\InetPub\wwwroot\PCREApplet.class`. The remaining attributes of the <applet> tag are optional. The `width` and `height` define the size of the bounding rectangle in which to display the applet's user interface display panel. The `name` and `id` attributes identify the applet to the browser.

In addition to the <applet> tag, you must supply at least two <param> tags that are necessary to enable the browser with the CORBA IIOP protocol.

The first <param> tag has the form

```
<param name=org.omg.CORBA.ORBClass
      value=com.visigenic.vbroker.orb.ORB>
```

and identifies a specific Object Request Broker to implement the IIOP protocol.

The second <param> tag has the form

```
<param name=ORBalwaysProxy value=true>
```

and ensures that all IIOP exchanges can be routed through firewalls. A full technical description of these terms is too lengthy to include here. You'll find additional information in the section entitled Applet Client Connection in Chapter 3. If you have difficulty in getting your applets to load, check with your System Administrator that your Parallel Crystal installation has been configured to run a component called the IIOP GateKeeper.

Remarks on Applet Programming

In the previous section we outlined a framework for constructing Parallel Crystal applet clients that uses the `init`, `start`, `stop and destroy` methods to manage the connection to the Report Server. The connection is maintained while the browser is displaying the HTML page containing the applet's tag, and is broken on navigating to a different page. If the browser returns to the applet's page, the connection is re-established but the client is connected to a new instance of the Report Engine.

It is possible to create an applet with different Report Server connection lifetimes by using the `init`, `start`, `stop`, and `destroy` methods in different ways.

For example, the following version of `init` creates a connection that only lasts until the end of the method.

```
public void init()
{
   PCREAppletClient client = null;
   try {
      client = PCREAppletClient();
      client.Connect();
      PCREEngine engine = client.OpenEngine();
      ...
      engine.Close();
   } catch ( PCREError e ) { e.Report(); }
   client.Quit();
}
```

This short-lived connection might be preferable if it is appropriate to perform all report processing before constructing the applet's user interface.

Another possibility is to establish the connection in `init` or `start` and then hold it open until the browser is terminated.

This can be accomplished with the framework in Example 2.5 on page 50.

```
public class PCREApplet extends java.lang.Applet
                        implements PCREConsole {

   private PCREAppletClient client;

   public void init()
   {
    try {
      client = new PCREAppletClient();
      ...
   }

   public void start()
   { /* empty */ }

   public void stop()
   { /* empty */ }

   public destroy()
   {
      ...
      client.Quit();
   }

}
```

Example 2.5

In this example, the client and its connection to the Report Server persist over the calls to `start` and `stop`. This means that the browser can navigate to other pages and continue to access the report on return to the applet page. This framework may be appropriate if the interaction with the Report Server takes place in a context that requires the display of several HTML pages or even several different applets.

A Java applet is run within your browser under the control of a Security Manager that imposes constraints on the external interfaces offered by the applet. In particular, your applet is only permitted remote communications with the network host running the web server and this in turn means that your Web Server and Report Server must be running on the same machine. This restriction applies to all Parallel Crystal "thin clients" that are implemented using Java applets.

In practice, it is possible to build browser-based Parallel Crystal thin clients that are not subject to this restriction by integrating with other technologies. For example, Microsoft's Active Server Pages technology described in Chapter 6 Using PCREAutomationServer allows HTML pages to be dynamically composed by a Web Server using information retrieved from a Report Server running on a different network host.

Chapter 3
Using the Java Client Package

The Parallel Crystal Java Client Package is a collection of classes providing a range of report management functions that you can access from Java applications or applets. The classes are arranged in functional groups shown in Figure 3.1, but are normally imported collectively into your program. Within each group, base classes are marked with an asterisk.

Error Handling	Console Management	Service Providers
*PCREError PCREFatalError PCREInternalError PCREClientError PCREGatewayError PCREServerError PCREAPIError	*PCREConsole PCREApplicationConsole	*PCREService PCREServer PCREGateway PCREConfigServer
Client Management	**PCRE API Helpers**	**PCRE API**
*PCREClient PCREApplicationClient PCREAppletClient	PCREGroupCondition PCREJobExportInfo PCREHTMLJobExportInfo PCREPDFJobExportInfo PCREPDFOutputOptions PCREMargins	PCREEngine PCREJob

Figure 3.1

This chapter describes the facilities provided by the Java Client Package including

- The section entitled Parallel Crystal Clients describes how to use the `PCREApplicationClient` and `PCREAppletClient` classes, and particularly how to specify the connection to your Report Server.

- Having established a connection to a Report Server, the section entitled Report Generation describes how to use the `PCREEngine` and `PCREJob` classes to perform report customization and generation. The methods of the PCREJob class are arranged in groups and illustrated with sample code fragments.

- When a report is stored in a file, it may be necessary to retrieve the file from the Report Server back to the machine hosting the client. The section entitled Report Retrieval describes facilities for automatic return of reports to the client.

- The section entitled Error Recovery describes how to use the error handling classes to detect and handle the various errors reported by Parallel Crystal.

- The section entitled Console Interface describes the abstract console interface that error-handling classes use to display error messages.

- The section entitled Parallel Crystal Configuration Server describes how clients can access information that describes the way in which their Parallel Crystal installation has been configured.

- The section entitled Parallel Crystal Load Balancer describes how Report Server hosts can be allocated to clients to ensure the load is balanced evenly or systematically across a group of servers.

- The section entitled Multi-threaded Clients describes how to construct multi-threaded clients that can perform report generation tasks concurrently.

- The section entitled Web Application Servers discusses a number of issues that arise when using the Parallel Crystal Java client from a Web Application Server.

The Parallel Crystal Java Client package is not a general-purpose class library in the sense of, for example, the Java Foundation Classes. Its primary purpose is to expose the Report Engine API in a manner that is convenient for Java Programmers, and to provide a small number of additional facilities that assist in integrating report processing into your application.

The sections that follow provide general information on how to use the PCRE Java Client. For full details of each class and its methods, you should consult the online Reference Documentation.

Parallel Crystal Clients

In this section, we describe how to create Parallel Crystal clients as Java applications or Java applets. Each client uses a set of "service provider" classes to access a corresponding set of Report Server programs that provide report generation, load balancing and connection management services. A client can connect to all services using a single API call, or can customize individual connections by setting properties through the service providers. In the following sections, we describe the how clients are created, how they connect to the Report Server, and how they can customize their connections. We begin with a short overview of the server programs running on the Report Server, and the services they provide.

Services for Clients

When a Parallel Crystal Report Server is started, it makes a range of services available to your client through special-purpose servers.

The special-purpose servers include

- The Gateway Server provides an initial point of contact for your client and is responsible for starting a Report Engine Server to generate your report. All clients share access to a single Gateway running on the Report Server. To connect to the Gateway, you must specify the Report Server host and (optionally) the name of the Gateway.

- The Report Engine Server provides report customization and generation facilities. The Report Engine is allocated to your client by the Gateway and is terminated when you disconnect from the Report Server. Report Engine Servers are not shared amongst clients.

- The Configuration Server allows clients to retrieve configuration information about their Parallel Crystal installation. For example you can obtain the name of the directory containing sample reports or the name of the directory to contain sample output. All clients share access to a single Configuration Server.

- The Load Balancer Server ensures that client connections to multiple Report Servers are allocated in a way that spreads the report-processing load evenly across the available server machines. Clients share access to the Load Balancer Server.

- The Custom Server is an extension to the Report Engine Server that is used to provide custom extensions to the Parallel Crystal Report Server.

Parallel Crystal Java clients gain access to these servers through the `PCREApplicationClient` and `PCREAppletClient` classes. When you call the client class's `Connect` method, it will establish a connection first to the Gateway server, and then to the allocated Report Engine Server. When you call the `Disconnect` method, these connections are broken and the Report Engine Server is terminated. The client classes also provide access to the service providers that can be used to fine tune the connection to the Gateway and control the operation of the Report Engine.

Parallel Crystal client programming always follows the same pattern: you instantiate your client, refine its configuration, and connect to the Report Server. Once you have finished your report customization you disconnect from the Report Server. You don't have to worry about deleting the instances of your clients because the Java language implementation does that for you. We'll explain these operational details in greater detail in the following sub-sections.

Application Client Instantiation

The `PCREApplicationClient` class allows stand-alone Java application clients to connect to the Parallel Crystal Report Server. The client has connect- and disconnect-methods and acts as a container for "service provider" classes that provide client-side access to Report Server services. You can install a console in the client to adapt the client's error reporting facilities to the user interface provided by your application.

A `PCREApplicationClient` is instantiated using a constructor, like this

```
PCREApplicationClient (PCREConsole console,
                       String host,
                       int port,
                       String[] args,
                       Properties props,
                       org.omg.CORBA.ORB orb)
        throws PCREClientError
```

The constructor argument `console` specifies an instance of a class implementing the `PCREConsole` interface. If you call an abbreviated constructor, then your client will be created with an instance of the default console class `PCREApplicationConsole`.

The constructor argument `host` specifies the Report Server to which your client will connect. It may take the form of either a symbolic host computer name, or a dotted host computer IP address.

For example:

```
PCREApplicationClient("example.host.com");
PCREApplicationClient("111.222.333.444");
```

The constructor argument `port` specifies the port at which the Gateway process provides its initial reference. When the client connection process occurs, the client obtains the initial reference to the Gateway from this port. The Gateway then provides the initial reference to all the other PCRE Services. This process used to involve obtaining the references to these objects from the OSAgent. The move toward more ORB independence in the Java Client mandated discontinuing usage of the OSAgent.

All client constructors throw a PCREClientError exception if the CORBA communication failed to initialize.

The constructor arguments `args` and `props` provide fine control over the connection of the client to the network. They are sometimes required when Report Servers are accessed across the Internet, or via firewalls, or when inter-operability is required between Parallel Crystal and one of the Java-oriented Web Application Servers. You should seek technical support if you believe you may have special operating requirements that may cause difficulty in connecting your client to the Report Server.

The constructor argument `orb` allows you to install a customized Object Request Broker in your client. If this argument is null, the client uses an ORB that is created using the `argv` and `props` arguments.

Applet Client Instantiation

The `PCREAppletClient` class allows Java applets to connect to the Parallel Crystal Report Server. The client has connect- and disconnect- methods and acts as a container for "service provider" classes that enable client-side access to Report Server services. You must install a console in the client to adapt error reporting to the user interface provided by your applet.

A `PCREAppletClient` is instantiated using a constructor, like this

```
PCREAppletClient (PCREConsole console,
                  String host,
                  int port,
                  java.applet.Applet applet,
                  Properties props,
                  org.omg.CORBA.ORB orb
        throws PCREClientError
```

All arguments have the same meaning as their equivalents in the `PCREApplicationClient` constructor, with the exception of `applet` that identifies the applet instance that contains your client. It is not possible to supply default values for either the console or the applet since each is specific to your applet. For example, the console you supply must adapt the abstract `PCREConsole` interface to the specific user interface requirements of your applet. It is not appropriate to supply a default instance of the character-based `PCREApplicationConsole` in this case.

Client Configuration

Each `PCREApplicationClient` or which `PCREAppletClient` instance acts as a container for a number of "service provider" classes that grant clients access to the various Parallel Crystal servers running on the Report Server machine. The service providers are created when you call the client's constructor, and are available through get property methods in the client. When the client calls the `Connect` method, connections are established between each service provider and its corresponding server. Prior to connection, it is possible to configure some service providers by invoking their set-property methods. Most commonly, this is used to control operation of the client's Report Engine Server.

The four service providers contained by the client classes are shown in Table 3.1. The function of each is summarized below.

Service Provider	Get Property	Purpose
`PCREGateway`	`GetGateway`	Access to PCRE Gateway Server
`PCREServer`	`GetServer`	Access PCRE Report Engine Server
`PCREConfigServer`	`getConfigServer`	Access to PCRE Configuration Server
`PCRECustomServer`	`getCustomServer`	Access to PCRE Custom Server

Table 3.1

PCREGateway

The `PCREGateway` class provides client access to the Parallel Crystal Gateway Server. The Gateway is responsible for allocating PCRE Report Engine Servers to clients.

PCREServer

The `PCREServer` class provides client access to the Parallel Crystal Report Engine Server. The Report Engine Server provides the report customization and generation functionality that you use to generate reports.

PCREConfigServer

The PCREConfigServer class provides client access to the Parallel Crystal Configuration Server. The Configuration Server provides information about the current configuration of your Parallel Crystal installation.

PCRECustomServer

The `PCRECustomServer` class provides special-purpose functionality to clients. Its function may vary depending upon your installation.

Using Get property Methods

The following fragment shows how the get property methods can be used to access the Gateway and Configuration servers:

```
PCREApplicationClient client = new PCREApplicationClient();
PCREGateway gateway = client.getGateway();
PCREConfigServer config = client.getConfigServer();
```

The `PCREApplicationClient` and `PCREAppletClient` classes are responsible for managing the life cycles of the service providers they contain. You should not attempt to create a service provider independently, or access its methods after you have called the client's `Disconnect` or `Quit` methods.

The `PCREGateway` and `PCREServer` providers are described briefly in the next two sections. The `PCREConfigServer` is described in the section entitled Parallel Crystal Configuration Server. If you don't plan to use these "advanced" features of Parallel Crystal, then skip directly to the sections entitled Application Client Connection and Applet Client Connection.

PCREGateway Properties

The `PCREGateway` provides client access to the Parallel Crystal Gateway Server running on the Report Server machine. The property methods allow you to specify the Report Server host and the name of the Gateway running on that host. They also allow you to specify the value of a timeout that controls the period of time that the Gateway will wait while starting a Report Engine Server. These properties are summarized in Table 3.2.

Property Name	Purpose
`setHost`	Set the Report Server host for the Gateway Server
`getHost`	Return the Report Server host running the Gateway Server
`setName`	Set the name of the Gateway Server
`getName`	Return the name of the Gateway Server
`setWait`	Set Gateway Server timeout when starting a PCRE Server
`getWait`	Return the Gateway Server timeout

Table 3.2

Here is a simple example in which the `setHost` property is used to specify the Report Server machine running the Gateway Server.

```
try {
   String hostName = HostNameDialog();
   client.getGateway().setHost(hostName);
   client.Connect();
   ...
} catch ( PCREError error ) {
   error.Report();
}
```

If you supply a Report Server host in a client constructor call, the constructor passes the host to the `PCREGateway` instance. If you omit the host, the `PCREAppletClient` constructor will pass the Web Server's host to the `PCREGateway` and the `PCREApplicationClient` will leave it unset. Either way, you can override these settings by using the `setHost` method as shown. When you call the client `Connect` method, the `PCREGateway` class is connected to the Gateway Server on the nominated host.

It only makes sense to use a set-property before you call the client's `Connect` method. If you call a set-property after connecting the client, the method will throw a `PCREClientError` exception.

```
try {
   client.Connect();
   String hostName = HostNameDialog();
   client.getGateway().setHost(hostName); // Exception thrown!
   ...
} catch ( PCREError error ) {
   error.Report();
}
```

PCREServer Properties

The `PCREServer` class provides client access to the Parallel Crystal Report Engine Server running on the Report Server machine. Table 3.3 summarizes the property methods, which allow you to supply or retrieve a string of command line arguments to be used when the Gateway Server starts the Report Engine Server. The command line arguments for the Report Engine Server are summarized in Table 3.4.

Property Name	**Purpose**
`setArguments`	Supply the command line argument string
`getArguments`	Return the command line argument string
`addArgument`	Append a command line argument to the string

Table 3.3

Argument	Purpose
`-verbose`	Switch on Report Engine Server tracing. This is useful if you are running Parallel Crystal on the desktop and you wish to see full tracing in the Report Engine's GUI.
`-folder directory`	Specify the Report Engine Server working directory. This lets you specify report files with path names relative to this directory.
`-client name`	Specify the name of the client. If you are running Parallel Crystal on the desktop, the Report Engine will display this string to help you identify your server amongst many.

Table 3.4

Here are two example fragments that specify the same arguments in equivalent ways:

```
/* Supply arguments in a single call. */
try {
   PCREServer server = client.getServer();
   server.setArguments("-client toby -folder C:\\MonthlyReports");
   ...
   /* The server will be started with arguments supplied. */
   client.Connect();
} catch ( PCREError error ) { error.Report(); }

/*
 * Add arguments in multiple calls. Argument values are
 * supplied through calls to user interface dialogs.
 */
try {
   PCREServer server = client.getServer();
   String name = ClientNameDialog();
   server.addArgument("-client " + name);
   String folder = ClientFolderDialog();
   server.addArgument("-folder " + folder);
   ...
   /* The server will be started with the arguments supplied. */
   client.Connect();
} catch ( PCREError error ) { error.Report(); }
```

In each case the argument string is supplied when the Report Engine Server is launched by the Gateway. This implies that you must know the values of the arguments before you issue the `Connect()` call. In the section Parallel Crystal Configuration Server we'll show how you can change the Report Engine Server's working folder after you have established the Report Server connection.

Application Client Connection

When you construct a Parallel Crystal Java application, you must first create a `PCREApplicationClient` class, then connect it to the Report Server, and then access the client's Report Engine.

Typically, this requires a code sequence such as

```
PCREApplicationClient client = null;

try {
   client = new PCREApplicationClient("example.host.com");
   client.Connect();
   PCREEngine engine = client.OpenEngine();
   ...
   engine.Close();
} catch ( PCREError e ) { e.Report(); }
client.Quit();
```

When the `Connect` method is called, the following events occur in sequence

- The client tries to establish a connection to a Gateway running on the Report Server. The Report Server is identified by the host name or host address supplied in the `PCREApplicationClient` constructor call, or in a call to the `PCREGateway` method `setHost`. In the example no host has been supplied, so the client will attempt to connect to any available Report Server host running a Gateway.

- If a connection to the Gateway is established, the client issues a request to the Gateway to start a Report Engine Server on the same host. The Gateway starts the Server with a command line that includes server arguments packaged in the request together with a server name that is unique for the Report Server. If the Server is started within a timeout period (usually 30 seconds), the Gateway returns the host and server name to the client.

- The client establishes a connection to the Report Engine Server using the name and host returned by the Gateway.

Once these connections have been established you can call the client's `OpenEngine` method to open the print engine in the Report Engine Server. The Report Engine is assigned exclusively to your client and cannot be accessed by other clients connected to the same Report Server. These connections last until you call the client's `Quit` or `Disconnect` methods.

If any of the preceding steps fail, the `Connect` method will throw an exception whose type and content identify the nature of the error. A `PCREGatewayError` indicates a failure to connect to the Gateway on the specified host, or a failure to launch the Report Engine Server correctly; a `PCREServerError` indicates a failure to connect to the Report Engine Server. Both exceptions are derived from the `PCREError` class so the easiest way to catch both is to specify `PCREError` in the catch-clause.

Connections to both the Gateway and Report Engine Servers are established on the basis of the host and name strings that are supplied through the service provider property methods. The host specifies a particular Report Server on your network, and the name identifies a particular instance of the Server on that host.

The Report Engine Servers are assigned unique names by the Gateway so that they can be identified to their clients. The Gateway Server is normally assigned the default name "pcre". A client is always connected to Gateway and Report Engine Servers on the same host.

Occasionally it may be more appropriate to make connections based on logical Server names rather than Report Server hosts. In order to do this, you must omit the host from the client constructor call and use the `PCREGateway`'s `setName` property method.

For example:

```
try {
   PCREApplicationClient client = new PCREApplicationClient(
      “example.host.com”);
   client.getGateway().setName("purple");
   client.Connect();
} catch ( PCREError e ) { e.Report(); }
```

This code will then try to connect your client to a Report Server running the Gateway named "`purple`" running on “`example.host.com`.”

You must ensure that the argument supplied to the `setName` method matches the name of a Gateway running on your Report Server. If not, the `Connect` method will throw a `PCREGateway` exception and the connection will fail. In the section entitled the Parallel Crystal Configuration Server we'll show how you can retrieve the Gateway name from the Report Server rather than hard-code it as a literal string.

Applet Client Connection

When you construct a Parallel Crystal Java applet, you must first create an instance of the PCREAppletClient class, then connect it to a Report Server, and then access the client's Report Engine.

For example:

```
PCREAppletClient client = null;
try {
   client = new PCREAppletClient(“example.host.com”)
   client.Connect();
   PCREEngine engine = client.OpenEngine();
   ...
   engine.Close();
} catch ( PCREError e ) { e.Report(); }
client.Quit();
```

The sequence of events that occurs when the `connect` method is called is

- The client tries to establish a connection to a Gateway running on the Report Server. If no host name is supplied in a call to the constructor or the PCREGateway's `setHost` method, the client will attempt to connect to a Gateway running on the same host as the Web Server.

- If a connection to the Gateway is established, the client issues a request to the Gateway to start a Report Engine Server on the same host. The Gateway executes the request by starting a uniquely named server and then returning the name and host to the client.

- The client establishes a connection to the Report Engine Server using the name and host returned by the Gateway.

Once the Gateway has been located, the applet client's `Connect` method proceeds in the same way as the application client. The connections remain until the client's `Quit` method is called.

Although the Java-code you write to connect an applet client or application-client is very similar, there are substantial differences at network-level that arise because of the restrictions imposed on applets by the browser's Security Manager. We'll devote the rest of this section to describing how applets get loaded into your browser and how the underlying CORBA framework establishes network connections to remote servers.

Applets are initially loaded using the <applet> tag in your HTML page. The `code` attribute specifies the name of a class with the `class` suffix omitted. The name may itself be a simple name such as `TravelApp` or a fully qualified name such as `com.acme.apps.TravelApp`. To locate the class, the browser adds a class-suffix and then searches an internal cache of loaded classes. This cache is partitioned into structured name spaces so the browser will search for the class `TravelApp.class` in the first case, and `com/acme/apps/TravelApp.class` in the second case[1]. If the class is not already loaded, the browser continues the search using the CLASSPATH of the host machine, and if that fails it issues an HTTP request to download the applet from the server machine.

An applet is downloaded by composing a URL for its class file from a code base and a code name. The code base specifies the Web server host and a sub-directory of its root directory. The code name is the name of the applet class.

[1] You can specify URLs using the forward-slash character '/'. Web-servers will automatically translate this to a back-slash character '\' on Windows-based machines.

If the HTML page containing the applet was retrieved from

`http://www.acme.com/travel.html`,

then the applet's class file will be retrieved from

`http://www.acme.com/TravelApp.class`

If the code attribute of the <applet> tag specified the full name com.acme.apps.TravelApp, then the applet's class file would be retrieved from

`http://www.acme.com/com/acme/apps/TravelApp.class`.

Sometimes it is convenient to organize a web server's root directory so that applets and HTML pages are held in different sub-directories. If this is the case, then the code base cannot be deduced from the original document URL and has to be specified using a `codebase` attribute in the <applet> tag. So if the TravelApp.class was actually held in the `classes` sub-directory of the Web Server root, then an <applet> tag with the following form would be required:

```
<applet code=TravelApp codebase=http://www.acme.com/classes >
```

Once your applet's class file begins to execute it will initiate searches for other classes in the PCRE package and the underlying CORBA framework. If you happen to have the Parallel Crystal Java Client installed on the same machine as the browser, then the classes will be loaded locally. Otherwise, they will be downloaded from the Web Server.

In attempt to reduce download time, some versions of Netscape Communicator have a CORBA framework pre-installed. However, we cannot guarantee that it will be compatible with the rest of the PCRE package.

So, we recommend that you always specify the applet parameter:

```
<param name=org.omg.CORBA.ORBClass
             value=com.visigenic.vbroker.orb.ORB>
```

This will force the client to download the CORBA framework distributed with the Parallel Crystal Java package and should ensure version compatibility in the client and the Report Server.

Once the CORBA framework and the core of the Java Client package have been downloaded, the client will try to connect to the Report Server using CORBA IIOP[1] protocol.

[1] IIOP stands for Internet Inter-ORB Protocol.

The IIOP connection will fail if either of the following is true

- The host specified for the Report Server is not the machine running the Web Server.
- The applet is running behind a firewall that permits only outgoing HTTP requests.

Running the Parallel Crystal IIOP GateKeeper on the Web Server machine can solve both problems. The GateKeeper acts as a staging post taking incoming IIOP requests and forwarding them to Report Server on the specified host. The GateKeeper also handles protocol restrictions by a technique called "HTTP tunneling" in which the client CORBA framework packages each IIOP request in an HTTP wrapper. When the GateKeeper receives the HTTP request, it removes the wrapper and forwards the original IIOP request to the server.

Normally, the CORBA framework tries to connect direct to the Gateway using IIOP. If the request fails, the framework will then re-try using the IIOP GateKeeper, and if necessary switching to HTTP protocol. In order to do this, you must first have the GateKeeper running on your Web Server, and you must tell the CORBA framework how to find it. When the GateKeeper is started, it creates a file called `gatekeeper.ior` whose contents allow the client CORBA framework to establish a GateKeeper connection. The CORBA framework retrieves the IOR file to the client by using a URL that you specify as an additional applet parameter.

```
<param name=ORBgatekeeperIOR
       value=http://www.acme.com/classes/gatekeeper.ior>
```

The URL looks very similar to the applet code base discussed above, and if you omit the `ORBgatekeeperIOR` parameter, the CORBA framework will attempt to retrieve the IOR file from the applet's code base directory by default.

You can reduce the connection time to the Report Server by having the client CORBA framework connect directly to the IIOP GateKeeper using the following applet parameter:

```
<param name=ORBalwaysProxy value=true>
```

The parameter name alludes to the fact that the IIOP GateKeeper is called a "proxy server" since it stands in for the real server on the remote machine.

The HTTP tunneling feature can be similarly enabled by the following parameter:

```
<param name=ORBalwaysTunnel value=true>
```

However, you should be aware that HTTP tunneling introduces the overhead of adding and removing HTTP wrappers to each IIOP request. Therefore you should be sure that you actually need this feature before enabling it.

Finally, we combine the various applet attributes and parameters into a complete <applet> tag. We assume that the HTML page was originally loaded from http://www.acme.com/pages/Travel.html but the applets are to be loaded from http://www.acme.com/classes/. The applet tag has the form:

```
<applet
   name=Travel width=400 height=600 code=Travel
   codebase=http://www.acme.com/classes >
   <param name=org.omg.CORBA.ORBClass
          value=com.visigenic.vbroker.orb.ORB>
   <param name=ORBalwaysProxy
          value=true>
</applet>
```

The `ORBgatekeeperIOR` parameter has been omitted so the file `gatekeeper.ior` should exist in the classes sub-directory of the Web Server root directory. The Parallel Crystal installation normally includes a GateKeeper IOR file at

```
C:\MobileApps\pcre\ior\gatekeeper.ior
```

So you can copy this file into your applet code base directory, or include an `ORBgatekeeperIOR` parameter of the form:

```
<param name=ORBgatekeeperIOR
    value=http://www.acme.com/MobileApps/pcre/ior/gatekeeper.ior >
```

Client ORB Management

Each application or applet client uses an Object Request Broker to manage the CORBA connections to the various PCRE servers running on the Report Server. When you create an application with multiple clients, possibly running in multiple threads, a single instance of the ORB is shared by all clients[1]. This strategy conserves resources, and can be important if the clients are deployed in a long-running Java Virtual Machine.

Suppose you create an application with multiple client instances such as:

```
PCREAppConsole console = new PCREApplicationConsole();
...
PCREApplicationClient client1 =
   new PCREApplicationClient(console, host1, args1, props1);
client1.Connect();
...
PCREApplicationClient client2 =
   new PCREApplicationClient(console, host2, args2, props2);
client2.Connect();
```

[1] Multi-threaded clients are discussed in the section at the end of this chapter.

If `client1` is instantiated first, its ORB is created using any relevant values supplied in `args1` and `props1`. Thereafter, this same ORB will be shared by `client2`, and any ORB-related values in `args2` and `props2` will be ignored. This situation applies independently of whether `client1` and `client2` are created on the same, or different threads.

Normally, sharing of ORBs between clients does not create problems, particularly if each client is connected to the same Report Server. However if the clients have distinct connection requirements that require conflicting argument values to be supplied to the ORB, then you must create a separate ORB for each client. You can do this with the static `createORB` method of the `PCREORBFactory` class.

```
PCREApplicationConsole console = new PCREApplicationConsole();
...
org.omg.CORBA.ORB orb1 =
   PCREORBFactory.createORB(args1, props1);
PCREApplicationClient client1 =
   new PCREApplicationClient(console, host1, args1, props1, orb1);
client1.Connect();
...
org.omg.CORBA.ORB orb2 =
   PCREORBFactory.createORB(args2, props2);
PCREApplicationClient client2 =
   new PCREApplicationClient(console, host2, args2, props2, orb2);
client2.Connect();
```

ORB-related arguments are supplied either as command line argument strings with the form:

```
-ORBname value
```

Or as property name/value pairs with the form:

```
ORBname value
```

Requirements such as these are, in fact, pretty unusual, and we advise you to consult with your System Administrator or our Technical Support Service before attempting to use customized ORBs.

Client Disconnection

The connections established between a Java application or applet client and the Report Server are severed when you call the client's `Quit` method.

- The client's connection to its Report Engine Server is severed and the server is terminated.
- Outstanding connections from the client to the Gateway are severed.

It is illegal to make a call to any of the `PCREEngine` or `PCREJob` methods after you have disconnected from the Report Server. Once you have called `Quit`, you must re-call `Connect` and acquire a reference to the new Report Engine Server.

Report Generation

In Parallel Crystal, the Report Customization API is exposed through the methods of the `PCREEngine` and `PCREJob` classes. These classes partition the API into two parts: methods that are report engine related and methods that are print job related. In this section we'll give a general overview of each class and provide some additional information on how to generate reports in particular formats. To understand the class interfaces in depth, you should consult the documentation supplied online with Parallel Crystal. Assuming installation on drive C, this is located in the directory `C:\MobileApps\pcre\docs`.

The following files should be present

- `JavaAPIDocumentation\index.html` is the root index to the PCRE package documentation. The documentation is formatted and cross-referenced according to the JavaDoc standard.

- `developr.hlp` is the Windows help file for the Seagate Crystal Reports Print Engine API.

You may need to consult both files to understand how the customization function is mapped to an equivalent method of the `PCREEngine` or `PCREJob` classes.

Accessing Report Engines and Print Jobs

Once you have connected your Java client to the Report Server, you get a reference to the Report Engine by calling the client's `OpenEngine` method.

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient("example.host.com")
   client.Connect();
   PCREEngine engine = client.OpenEngine();
   /* run print jobs */
   ...
   engine.Close();
} catch ( PCREError e ) { e.Report(); }
client.Quit();
```

Your client only has access to a single Report Engine Server so successive calls to `OpenEngine` will return the same value. When you are finished you should call the `Close` method to release resources in the Report Engine Server. If you call `OpenEngine` when your client is not connect to a Report Server, it will throw a `PCREClientError` exception.

The most important methods of the `PCREEngine` class are summarized in Table 3.5 on page 68.

Method	Purpose

`OpenJob`
`OpenSubReport`
`PrintReport`
`ParallelPrintReport`
	Open a print job
	Open a sub-report
	Start a report run
	Start a parallel report run

`LogOnServer` `LogOffServer`	Log on to database server Log off database server
`Open` `Close`	Open a print engine Close print engine

Table 3.5

The PCREEngine class allows you to produce your report in one of three ways

- You can generate the report directly using the `PrintReport` method. The single argument to the method specifies the name of the report file on the Report Server. No report customization is possible with this method.

- You can open a print job with the `OpenJob` method. The method takes a single argument specifying the name of the report file on the Report Server, and returns a reference to an instance of the `PCREJob` class representing the print job. You can call methods of `PCREJob` to customize the report and then generate it by calling the `Start` method.

If your report requires access to a database server, then you must call the `LogOnServer` method before calling any of the methods `OpenJob`, `PrintReport`, or `ParallelPrintReport`. Since you only have access to a single report engine, it is only possible to open a single database connection at a time. You should call the corresponding `LogOffServer` method when your report has been generated.

The `Close` method closes the engine and releases resources within your Report Engine Server. You cannot subsequently call any of the report management methods unless you re-open the engine with another call to `Open`.

The following code fragment shows some typical calls to the PCREEngine methods:

```
client.Connect();
PCREEngine engine = client.OpenEngine();

/* Generate the Box Office report. */
engine.PrintReport("Box.rpt");

/* Customize Box Office report. */
PCREJob job = engine.OpenJob("Box.rpt");
job.OutputToPDF
       ("C:\\MobileApps\\pcre\\SampleOutput\\Box.pdf", null);
job.Start();
job.Close();
```

Customizing Print Jobs

The PCREJob class provides methods to perform a wide range of report customization tasks. Each time you call the PCREEngine's OpenJob method, you create a PCREJob instance representing a print job opened on the nominated report file. You can have multiple print jobs opened simultaneously and you may customize and run each job independently of the others. Each call to the PCREJob's Start method will run the job and generate a copy of the report. When you are finished with the print job you should call the PCREJob's Close method to release resources in the Report Engine Server.

In the following sections we present groups of related PCREJob methods together with sample code fragments illustrating typical calls. For brevity we omit the surrounding try-catch block, but you must include it in real applications. For more realistic examples you should consult the sample code distributed with Parallel Crystal.

Print Job Execution

The following group of methods controls the execution of a print job and returns its status.

Start	Close	CloseSubReport	Cancel

For example:

```
PCREJob job = engine.OpenJob("MyReport.rpt");
/* Customize job ... */
job.Start();
job.Close();
```

You should always call Close or CloseSubReport after the report has been generated to release resources in your Report Engine Server.

Print Job Details

The following group of methods allows you to manipulate details of your report.

```
GetReportTitle    SetReportTitle  SetPrintOptions  HasSavedData
DiscardSavedData  GetPrintDate    SetPrintDate
```

For example:

```
int [] date = new int[3];
date[0] = year; date[1] = month; date[2] = day;
job.SetPrintDate(date);
job.SetReportTitle("Monthly Sales: January – March");
job.Start();
```

Print Job Outputs

The methods in this group are amongst the most important in PCREJob and allow you to specify the format and destination of your report.

```
OutputToPrinter    OutputToPDF    ExportTo
```

We have already given simple examples of calls to OutputToPrinter and OutputToPDF in Chapter 2. In this section we'll show how to use `ExportTo` to generate reports in different output formats and for various storage media.

The `ExportTo` method takes a single parameter that is a reference to a helper class called `PCREJobExportInfo`. This class encapsulates information about the format and the destination of the report.

For example:

```
public class PCREJobExportInfo {
   public String formatDLLName;
   public int formatType;
   public Object formatOptions;
   public String destinationDLLName;
   public int destinationType;
   public Object destinationOptions;
}
```

The report format is defined by the `formatDLLName` and `formatType` members. The DLL name specifies a DLL which must be supplied with the Parallel Crystal Installation and which generates the report in the appropriate format. Some formats have minor variations and so the `formatType` name is used to hold a numeric value specifying sub-typing information. For example, there are five variants of Excel format available, represented by constants called `UXFXls2Type`, `UXFXls3Type`, etc. Some output formats take additional format-dependent data, and these are supplied through a variety of helper classes called `UXF_xxxOptions`. Each options helper has a constructor that makes it easy to instantiate the class with the appropriate additional information.

The report destination is supplied in a similar way through the `destinationDLLName`, `destinationType` and `destinationOptions` members. The destination DLL specifies a DLL that is responsible for dispatching the report. The `destinationOptions` member supplies additional destination-dependent data, and these are supplied through helper classes called `UXD_xxxOptions`. For example, if you specify the `destinationDLLName` `"u2ddisk"` which saves a report to a disk file, then the `destinationOptions` member should be an instance of the class `UXD_DiskOptions` that holds the path name of the file to be created.

In Table 3.6 on the next page, we list the possible values for `formatDLLName`, their corresponding sub-type values and the format-dependent helper class name. Then entry "none" means that the `formatOptions` member should be set to `null`.

DLL Name	Format Sub-type	Format Options
u2fcr.dll	UXFCrystalReportType	None
u2fdif.dll	UXFDIFType	UXF_DIFOptions
u2frec.dll	UXFRecordType	UXF_RecordStyleOptions
u2frtf.dll	UXFRichTextFormatType	None
u2fsepv.dll	UXFCommaSeparatedType UXFTabSeparatedType UXFCharSeparatedType	UXF_CommaTabSeparatedOptions UXF_CommaTabSeparatedOptions UXF_CharSepatedOptions
u2ftext.dll	UXFTextType UXFTabbedTextType UXFPaginatedTextType	UXF_PaginatedTextOptions
u2fwks.dll	UXFLotusWksType UXFLotusWk1Type UXFLotusWk3Type	None
u2fwordw.dll	UXFWordWinType	None
u2fdoc.dll	UXFWordDosType UXFWordPerfectType	None
u2fqp.dll	UXFQP5Type	None
u2fxls.dll	UXFXls2Type UXFXls3Type UXFXls4Type UXFXls5Type UXFXls5TabType	UXF_XlsOptions
u2fodbc.dll	UXFODBCType	UXF_ODBCOptions
u2fhtml.dll	UXFHTML3Type	UXF_HTML3Options
u2fmaspdf.dll	UXFPDFDistillerType	UXF_PDFOptions

Table 3.6

In the following example, we use the `ExportTo` method to generate a file in Excel format and save it to disk:

```
PCREJob job =
   engine.OpenJob("C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");

PCREJobExportInfo info = new PCREJobExportInfo();
info.formatDLLName = "u2fxls.dll";
info.formatType = PCREJob.UXFXls4Type;
info.formatOptions = new UXF_XlsOptions(false);
info.destinationDLLName = "u2ddisk.dll";
info.destinationType = PCREJob.UXDDiskType;
info.destinationOptions = new UXD_DiskOptions(
     "C:\\MobileApps\\pcre\\SampleOutput\\Box.xls");
job.ExportTo(info);

job.Start();
```

Notice that you must call `Start` after `ExportTo` to actually generate the report.

Table 3.7 below enumerates the possible settings for the `destinationDLLName` and the corresponding values for the `destinationType` and `destinationOptions`.

DLL Name	Destination Sub-type	Destination Options
u2ddisk.dll	UXDDiskType	UXD_DiskOptions
u2dmapi.dll	UXDMAPIType	UXD_MAPIOptions
u2dpost.dll	UXDExchFolderType	UXD_PostFolderOptions

Table 3.7

The following example shows how to mail a report using the MAPI mailer server interface:

```
PCREJob job =
   engine.OpenJob("C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");

PCREJobExportJobInfo info = new PCREExportJobInfo();
info.formatDLLName = "u2ftext.dll";
info.formatType = PCREJob.UXFTextType;
info.formatOptions = null;

info.destinationDLLName = "u2dmapi.dll";
info.destinationType = PCREJob.UXDMAPIType;
UXD_MAPIOptions options = new UXD_MAPIOptions();
options.toList = "director@reporthost.com";
options.ccList = "";
options.subject = "Box Office Report";
options.message = "Here is your Box Office Report:\r\n\n";
info.destinationOptions = options;

job.ExportTo(info);
job.Start();
```

Early releases of Parallel Crystal use the `ExportTo` method to generate reports in PDF. When the Report Engine Server receives export information specifying `UXFPDFDistillerType` as the output type, it generates the report in PostScript and then invokes the Adobe Acrobat Distiller utility to convert the PostScript into the final PDF file. A helper class called `PCREPDFJobExportInfo` generates the required `PCREJobExportInfo` by specifying the output file in a simple constructor call. The following example shows how to use this class:

```
PCREJob job =
   engine.OpenJob("C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");

PCREPDFJobExportInfo info =
   new PCREPDFJobExportInfo
       ("C:\\MobileApps\\pcre\\SampleOutput\\Box.pdf");
job.ExportTo(info);
job.Start();
```

Although PDF generation using Acrobat Distiller is slower than direct generation using the `OutputToPDF` method, it is retained for backwards compatibility.

In Example 2.3 on page 39 in Chapter 2 we showed how to generate HTML reports using the `PCREHTMLJobExportInfo` helper class. This class is an extension of the `PCREJobExportInfo` that automatically sets the formatting DLL to `u2fhtml.dll` and the destination DLL to `u2ddisk.dll`.

There are also convenience classes for Rich Text (`PCRERTFJobExportInfo`), Microsoft Word (`PCREWordJobExportInfo`), and Microsoft Excel (`PCREExcelJobExportInfo`).

Formula Manipulation

The following group of methods allows you to retrieve and adjust the formula fields and record or group selection formulae in your report.

```
GetNFormulas             GetNthFormula        GetFormula
SetFormula               CheckFormula         GetSelectionFormula
SetSelectionFormula      GetGroupSelectionFormula
SetGroupSelectionFormula CheckGroupSelectionFormula
```

Formula fields are accessed by an index that ranges from `0` to `(GetNFormulas()-1)`. The retrieval methods return a string so you must allocate a `StringBuffer` to hold it.

The following fragment returns all the formula fields in a report:

```
int nFormulas = job.getNFormulas();
StringBuffer nameBuffer = new StringBuffer(32);
StringBuffer formulaBuffer = new StringBuffer(128);
for ( int i = 0; i < nFormulas; i++ ) {
   nameBuffer.setLength(0);
   formulaBuffer.setLength(0);
   job.GetNthFormula(i, nameBuffer, formulaBuffer);
   System.out.println("n = " + i + " " +
                      "name = " + nameBuffer + " " +
                      "formula = " + formulaBuffer);
}
```

The methods `GetFormula` and `SetFormula` allow formula fields to be accessed by name rather than index. The remaining methods have simple get- and set- methods.

Parameter Field and Stored Procedure Manipulation

The following group of methods allows you to manipulate the parameter fields and stored procedure parameters in your report:

```
GetNParameterFields  GetNthParameterField  SetNthParameterField
GetParameterField    SetParameterField
```

Parameter fields are accessed by an index value ranging from `0` to `(GetNParameterFields() – 1)`. The `GetNthParameterField` method accesses the n'th field and returns its attributes in a helper class called `PCREParameterFieldInfo`.

The following code fragment sets the value of a report's parameter:

```
PCREParameterFieldInfo pinfo = job.GetNthParameterField(0);
PCREValueInfo value = pinfo.CurrentValue;

value.viString("Parameter value");
pinfo.CurrentValue=value;
pinfo.CurrentVlueSet = true;

job.SetNthParameterField(0, info);
```

The `GetParameterField` and `SetParameterField` methods allow more convenient access by using the name of the field rather than its index value. The most important thing to note is the line `pinfo.CurrentValueSet=true`. If this isn't done, the Report Server will seem to hang, because the Crystal Engine will display a dialog box on the server, asking for the value of the parameter.

Database Table Manipulation

The following group of methods allows you to access database tables within your report:

```
GetNTables           GetNthTableType        GetNthTableLocation
SetNthTableLocation  GetNthTableSessionInfo GetNthTableLogonInfo
SetNthTableLogonInfo TestNthTableConnectivity
```

Tables are accessed by an index value ranging from 0 to `(GetNTables() –1)`. The following fragment shows how to log the first table onto the database.

```
//this value tells the server whether or not to try to
//  use this logon information for the other tables in the
//  report
boolean propagate = true;

PCRELogOnInfo info = job.GetNthTableLogOnInfo(0);
info.Password = "password";
job.SetNthTableLogOnInfo(0,info,propagate);
```

If you look at the `PCRELogOnInfo` structure in the API reference, you will notice that it has several other fields. Here, we only set the `Password` field, because the call to `GetNthTableLogOnInfo` fills in the other fields (they are stored with the report).

The methods `GetNthTableSessionInfo` and `GetNthTableLocation` use similar helper classes `PCRESessionInfo` and `PCRETableLocation` to return their results.

SQL Queries

The following group of methods allows you to access the SQL Query string associated with your report:

```
GetSQLQuery    SetSQLQuery
```

The following fragment shows how to retrieve and reset the SQL query. You must allocate a StringBuffer to hold the query string.

```
StringBuffer queryBuffer = new StringBuffer(128);
job.GetSQLQuery(queryBuffer);
System.out.println("SQL Query: " + queryBuffer);
job.SetSQLQuery("SELECT * FROM CUSTOMER");
```

Sort Field Manipulation

The following group of methods allows you to retrieve and set the sorting fields for the records and groups within your report:

```
GetNSortFields        GetNthSortField         SetNthSortField
DeleteNthSortField    GetNGroupSortFields     GetNthGroupSortField
SetNthGroupSortField  DeleteNthGroupSortField
```

Sort fields are accessed by an index that ranges from `0` to `(GetNSortFields() – 1)`. GetNthSortField returns the value of the n'th sort field in a pre-allocated `StringBuffer`. The following fragment prints all the sort fields in a report:

```
int nFields = job.GetNSortFields();
for ( int i = 0; i < nFields; i++ ) {
   StringBuffer sortBuffer = new StringBuffer(128);
   job.GetNthSortField(i, sortBuffer);
   System.out.println("Sort field " + i + ": " + sortBuffer);
}
```

Sections and Groups

The following group of methods allows you to access the sections and groups within your report:

```
GetNSections    GetNGroups    GetGroupCondition  SetGroupCondition
GetSectionCode
```

Sections are accessed by an index that ranges from `0` to `(GetNSections() – 1)`. The type of a section is retrieved by `GetSectionCode()`, and the condition associated with a group is retrieved as the value of a `PCREGroupCondition` helper class.

The following fragment returns the group conditions for all sections in a report:

```
int nSections = job.GetNSections();
for ( int i = 0; i < nSections; i++ ) {
   int section = job.GetSectionCode(i);
   PCREGroupCondition cond = job.GetGroupCondition(section);
   /* Process condition ... */
}
```

Area and Section Formatting

The following group of methods allows you to adjust the areas and sections in your report:

```
GetSectionFormat            SetSectionFormat
GetAreaFormat               SetAreaFormat
GetMinimumSectionHeight     SetMinimumSectionHeight
GetSectionFormatFormula     SetSectionFormatFormula
GetAreaFormatFormula        SetAreaFormatFormula
```

Sections and areas are accessed by a section code returned by the `GetSectionCode` method. The formatting details of each section or area are specified in the attributes of a `PCRESectionOptions` helper class.

The following fragment shows how the formatting of the n'th section is retrieved and reset:

```
int section = job.GetSectionCode(n);
PCRESectionOptions options = job.GetSectionFormat(section);
/* adjust formatting options ... */
job.SetSectionFormat(section, options);
```

Sub report Management

The following group of methods allows you to access sub-reports within a given report:

```
OpenSubReport               CloseSubreport
GetNSubreportsInSection     GetNthSubreportInSection
```

Sub reports are accessed by `GetNthSubreportInSection` that returns a `PCRESectionInfo` helper class containing the name of the sub-report. You can open a job for the sub-report by passing the name to the `OpenSubreport` method of the `PCREEngine` class.

The following fragment iterates through all the sub-reports in the n'th section:

```
int section = job.GetSectionCode(n);
int nSubReports = job.GetNSubreportsInSection(section);
for ( int i = 0; i < nSubReports; i++ ) {
   PCRESubreportInfo info = GetNthSubreportInSection(section, i);
   PCREJob subReport = engine.OpenSubreport(job, info.name);
   /* customize sub-report ... */
   subReport.CloseSubreport();
}
```

Graph Manipulation

The following group of methods allows you to manipulate graphs and charts in your report:

```
GetGraphType  GetGraphData  GetGraphText GetGraphOptions SetGraphType
SetGraphData  SetGraphText SetGraphOptions
```

Graphs are located by a section code returned by the `GetSectionCode` method. Each section can contain multiple graphs that are numbered from 0 upwards. The `GetGraphOptions`, `GetGraphData` and `GetGraphText` methods respectively return the required information in the helper classes `PCREGraphOptions`, `PCREGraphDataInfo` and `PCREGraphTextInfo`. `GetGraphType` returns an integer-valued type code identifying the type of the graph. The corresponding set methods use the same helper classes when resetting a property of the graph.

The following fragment shows how the formatting options and title string for a graph in the n'th section are retrieved and reset.

```
int section = job.GetSectionCode(n);
PCREGraphOptions options = job.GetGraphOptions(section, 0);
options.verticalBars = true;
job.SetGraphOptions(section, 0, options);
PCREGraphTextInfo info = job.GetGraphText(section, 0);
info.graphTitle = "New Title";
job.SetGraphText(section, 0, info);
```

Data Object Reporting

Overview

Data Object Reporting allows users of Parallel Crystal to have more control over the data used in their reports. Until now, users could only report on data that was in files or databases. Web Application Server users had to allow Parallel Crystal to bypass some of the access control and database connection pooling features of their Application Server.

With Data Object Reporting, Parallel Crystal users can push data from data objects into Parallel Crystal servers and generate reports. From simple data objects, which perform limited filtering, to advanced data objects which filter, perform access control, contain not only stored data but also calculated data, and pass the data through a layer of complex business logic, Data Object Reporting allows users to specifiy these data objects as their data sources for their reports.

Using design time utilities, a report file that contains only the data access information is created. Then, using the Crystal Reports Designer, the empty report file is used as a base to add fields, charts, graphs, text, etc. to the report design. At run time, the data object is specified as the data source for the report template, and the report is generated.

Preparation

Creating a report for use with Data Object Reporting is much like designing a report for any other Parallel Crystal application. The result is still a Crystal Report template (.rpt) file that is used by Parallel Crystal. The main difference is that the DOR report accesses data differently. Therefore, the creation of the report template for DOR requires some special consideration, extra analysis and utility work, as well as some system preparation.

Preparation for Creating DOR Reports includes

- Installing DOR to RPT Utility.
- Analyzing the Data Source.

Steps for Creating a DOR Report

- Run DOR Design Utility to generate inital DOR file.
- Run DOR to RPT Utility to create a Crystal Reports template (.RPT) from the DOR file.
- Final formatting using Crystal Reports Designer.

Installing DOR to RPT Utility

The DOR to RPT Utility should be installed on the machine that will be used to design DOR reports. This may be a different machine than the one used as a Client to run reports.

This utility is installed using an automatic install program called DORReportUtility.exe. This program is placed into the \MobileApps\PCRE\DataObjectReporting folder during a Parallel Crystal installation.

Determine which computer will be used to design DOR reports and copy the DORReportUtility.exe onto that machine. Then, run the DORReportUtility.exe install program and the utility will be automatically installed.

Analyze the Data Source

It is important to identify and analyze the data source before attempting to use DOR. The process of preparing report templates requires knowledge of the type of data source involved.

If your data source is a database then you can use the ResultSet adapter. If not, or your data is being generated, calculated, or exists in an object or a set of objects then you need to implement the DORDataSet interface.

Implementing the DORDataSet Interface

The DORDataSet interface is used by the PCRE Java Client to examine a data object.

The interface has two main parts:

- A set of data type constants, which provide a way of specifying what data types an object contains, in a manner that is compatible with Crystal Reports.
- Methods to retrieve data from a data object, which provide a means of describing a data object and giving access to the various data types it contains.

Implementing the DORDataSet interface is similar to implementing any other interface in Java. Please see a java reference for the language specifics of this process.

The methods of the DORDataSet interface can be devided into 3 groups:

- Descriptive
- Data Retrieval
- Administrative

Descriptive Group

```
getColumnCount()            :  returns the number of columns in your
                            :  data set

getColumnDisplaySize(int)   :  if the column is a string or similar
                            :  displays the width of the field.

getColumnLabel(int)         :  returns the name of the column

getColumnType(int)          :  returns the data type constant for that
                            :  row
```

Data Retrieval Group

Each of these methods attempts to return the data element in the current row and the column specifed by the int argument as a specific data type. If the specified element cannot be retrieved as the given type then an exception should be thrown. For example, if getInteger was requested on a data element that was a String, since String data can't be represented very well as an integer, then that method should throw an exception in that case.

```
getBlob(int)

getBoolean(int)

getByte(int)

getCurrency(int)

getDate(int)

getDouble(int)

getInteger(int)

getMemo(int)

getShort(int)

getString(int)
```

Administrative Group

```
close()     This method will be called by PCRE after retrieving the
            data from the data object.  Use this method to close any
            open connections and free any resources as necessary.

next()      This method advances the row pointer to the next row of
            data.  It is assumed that the pointer points to BEFORE the
            first row when the object is initialized (ie PCRE will call
            next() before attempting to retrieve any data).  When the
            pointer is past the last row, this method should return
            false.
```

General Suggestions for implementing the data interface

- Have the data object itself implement the interface. This means fewer classes to keep track of, but it will mean that the data object will have more methods, and could be more complex.

- Write a data object descriptor for the data object. The data object descriptor implements the interface to describe a single data object. This means more classes, but each class is more specific, a data object, and its descriptor.

- Write a generic data object descriptor that can be used to describe any number of data objects. Perhaps using Java's reflection capabilities to call get-type methods with the data objects. This means having fewer data object descriptors to maintain, but also means that each data object will have to follow a common pattern.

Using ResultSet interface

The DORDataSet interface is closely modeled after the java.sql.ResultSet interface. Parallel Crystal has its own interface, which is simpler than using the java interface. The ResultSet interface provides additional capabilities, which were unnecessary for DOR and make implementing the interface more difficult. For those users who either prefer to code to the ResultSet interface, or who use JDBC to retrieive data, we have provided an adaptor class so that the ResultSet interface can be used with DOR.

Using the ResultSet Adapter

If your data source is a JDBC database, then the result of your query will be returned as an object that implements the java.sql.ResultSet interface. Since DOR expects its input to be of the form of the DORDataSet interface the adaptor DORSQLResultSetAdapter was created. This class implements the DORDataSet interface and takes a java.sql.ResultSet object in its constructor. The datatype, and method mapping of the adapter are listed in the appendix. Below is an example of use. See the javadoc comments in the Java API Quick Reference for additional details on the adapter.

```
//get ResultSet from jdbc connection
// resSet is the ResultSet

//Instantiate the adapter and pass it the ResultSet
DORSQLResultSetAdapter adapt = new DORSQLResultSetAdapter(resSet);

//do other PCRE calls

//make call specifying adapted ResultSet as data source for report
aJob.SetPrivateData((DORDataSet)adapt);

//continue with PCRE calls
```

DOR Design Utility

The DOR Design Utility is included in the PCREJavaClient.jar. This utility creates a DOR file by evaluating a Data Object. Running this utility results in the creation of a DOR file in the format that the **DOR to RPT Utility** expects. The correct syntax for creating a DOR File from a Data Object is listed below.

Running the DOR Design Utility

From a command line:

```
java com.mobileapps.dor.DorDesignUtil ResultSetClassName
MetaDataClassName OutputFileName
```

Running as

java com.mobileapps.dor.DorDesignUtil

Will cause a usage message to be printed to the screen.

From within a program:

```
//example code
DORDataSetImpl dorDS = DORDataSetImpl;
DorDesignUtil dUtil = new DorDesignUtil();
DUtil.outputDorFile((DorDataSet)dorDS, "filename");
```

DOR to RPT Utility

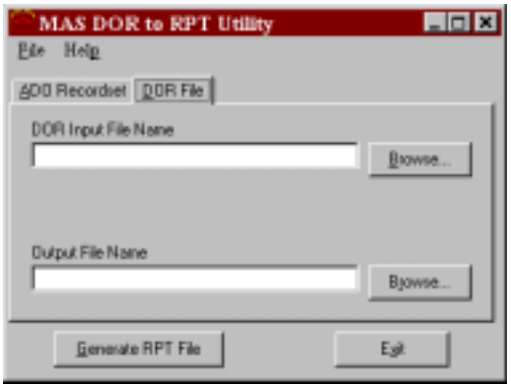
The **DOR to RPT Utility** creates a Crystal Reports (.rpt) template by evaluating the DOR (.dor) file created by the **DOR Design Utility**. Running the **DOR to RPT Utility** results in a pre-loaded skeleton of a report template with links to the Data Object, but whose formatting should be completed using the Crystal Reports Designer. The instructions for the use of this utility are listed below

Note: This utility is installed by running the program:

```
\MobileApps\Pcre\DataObjectReporting\DORReportUtility.exe
```

Running the DOR to RPT Utility

1. From the Start Menu, choose >>Programs >>Parallel Crystal .>>DOR Report Utility. The DOR to RPT Utility dialog box appears.



The DOR Input file is the file created using the DOR Design Utility. The Output File Name is the name of the report file to be created.

2. If the data source is an ADO Recordset, choose the ADO Recordset tab.

If you are using DOR with the Automation Server, then your data source is likely an ADO Recordset. Enter the Source string and the Connection text string (like you would in the code to create such a recordset) in the appropriate boxes. The Output File Name is the name of the report file to be created.

Hitting the Generate RPT File button will cause a blank report skelton to be created and a dialog box similar to the one below will be shown

Final Report Design

Final formatting of the report template should be completed using the Crystal Reports Designer.

Running a DOR Report

Running the DOR Report Template. This code snippet instantiates a data object that implements the DORDataSet interface, connects to a report server, sets the data object as the data source for the report, generates the report as PDF and disconnects from the report engine. The important line of code here is in bold.

```
PCREApplicationClient client = null;

// YourDataObject implements the DORDataSet interface
DORDataSet dataObj = (DORDataSet) new YourDataObject();

try
{
       //connect client to report server
       client = new CREApplicationClient("reportserver.host.com");
       client.Connect();

       //create engine and job objects
       PCREEngine engine = client.OpenEngine();
       //open the report that you created in the previous steps
       PCREJob job = engien.OpenJob("report_filename.rpt");

       //set your data object as the data source
       job.SetPrivateData(dataObj);

       //set the output type and start the report generation
       job.OutputToPDF("output_filename.pdf",null);
       job.Start();

       //close down and clean up
       job.Close();
       engine.Close();
}
catch(PCREError ex)
{
       //do some error handling here
}
finally
{
       //disconnect from report server
       if (client != null) client.Quit();
}
```

Tips for Using DOR

Limits on Data Object Size

DOR in PCRE 2.3 and 2.4 supports small to medium sized data objects (under 25,000 rows of data). Using DOR with larger data sets can cause excessive memory usage, which can lead to system instability. Actual limits of data object size will depend on hardware configuration, user traffic, and resources available to the program. This is an issue on the machine running the PCRE Java client, not the PCRE server machine.

DOR Frequently Asked Questions

1) ***DOR supports the java.sql.ResultSet interface through the use of an adapter. Does this mean that Data Object Reporting users have to be using a database?***

No. Although DOR supports that interface, there is no underlying reliance on databases, JDBC, or ODBC. We provide support for that interface to make DOR easier to use for JDBC users.

2) ***I use software that can collect data from a variety of sources (databases, data objects, etc) into a central data object. Can I use DOR?***

Yes. As long as you can use the DORDataSet interface to describe this central location, you can use DOR to report on that information.

3) ***DOR interests me, but I need to use one of Parallel Crystal's non-Java client libraries (C, C++, Active X), what can I do?***

With PCRE 2.4 our Automation Server supports Microsoft ADO Recordset objects as a DOR data source. See the Automation Server documentation for additional details.

4) ***I'm a SilverStream, Weblogic, NetDynamics, Websphere, etc, Web Application Server (WAS) user who uses Parallel Crystal, can I use DOR?***

As long as your WAS environment will allow you to create an object that implements the DORDataSet interface and describes your data source, you can use DOR with Parallel Crystal.

5) ***I don't use a Web Application Server, can I use DOR?***

DOR has no built in dependencies on WAS's. If you can implement the interface and use Parallel Crystal, then you can use DOR.

6) ***I don't use any data objects, all my data exists in a database. However I like the other aspects of DOR (access control, database connection pooling) can I still use DOR?***

Yes. In fact, if your database will return data in a manner compliant with either java.sql.ResultSet or DORDataSet interfaces, then using DOR will be very easy for you.

7) ***Can I use DOR to report on real-time data?***

DOR can be used to create a report on a snapshot of a real-time data source. This means that the data in the report will only change if you generate the report again.

8) ***Does DOR have any underlying depencies on environment or data source?***

As long as you can use Parallel Crystal in your environment, and you can use the interface to describe and provide access to your data source, then you can use DOR.

Datatype Mapping Reference

Crystal	DORDataSet	java.sql.ResultSet
Currency	CURRENCY	java.sql.Types.DOUBLE
Boolean	BOOLEAN	java.sql.Types.BIT
Date	DATE	java.sql.Types.DATE java.sql.Types.TIME java.sql.Types.TIMESTAMP
Decimal	DOUBLE	java.sql.Types.FLOAT java.sql.Types.REAL java.sql.Types.DOUBLE java.sql.Types.DECIMAL java.sql.Types.NUMERIC
Number	INTEGER	java.sql.Types.INTEGER java.sql.Types.BIGINT (w/ truncation)
Blob	BLOB	java.sql.Types.BINARY java.sql.Types.VARBINARY java.sql.Types.LONGVARBINARY
short	SHORT	java.sql.Types.SMALLINT
Byte	BYTE	java.sql.Types.TINYINT
Memo	MEMO	java.sql.Types.LONGVARCHAR
string	STRING	java.sql.Types.CHAR java.sql.Types.VARCHAR java.sql.Types.NUMERIC java.sql.Types.DECIMAL

The behavior for types NUMERIC and DECIMAL depend on the settings by setNumeric andDoubleConversionType. Use the constants REPRESENT_AS_DOUBLE_OR_CURRENCY or REPRESENT_AS_STRING. See the PCRE Java Quick Reference for additional details.

Types not listed (which includes types introduced in Java 2) will cause an exception to be thrown by the adapter. However we have done testing with Access, SQL Server and Oracle and DOR handles all the data types returned by their drivers.

Method Mapping Reference

DORDataSet	ResultSet (types listed are java.sql.Types.*)
close	close
getBlob	getBinaryStream for LONGVARBINARY get Bytes for BINARY, VARBINARY
getBoolen	getBoolean
getByte	getByte
getColumnCount	getMetaData.getColumnCount
getColumnDisplaySize	getMetaData.getColumnDisplaySize
getColumnLabel	getMetaData.getColumnLabel
getColumnType	(custom mapping, see table above)
getCurrency	getDouble for FLOAT, REAL, DOUBLE getBigDecimal for DECIMAL and NUMERIC
getDate	getDate for DATE getTime for TIME getTimeStamp for TIMESTAMP
getDouble	getDouble for FLOAT, REAL, DOUBLE getBigDecimal for DECIMAL and NUMERIC
getInteger	getInt
getMemo	getString
getShort	getShort
getString	getString
Next	next

Error Recovery

The Parallel Crystal Java Client package provides a group of classes for handling and reporting errors. These classes are arranged in the hierarchy shown in Figure 3.2. All classes are derived from `PCREError` but `PCREGatewayError`, `PCREServerError` and `PCREAPIError` are additionally derived from `PCREServiceError` in order to group errors that originate in the Report Server.

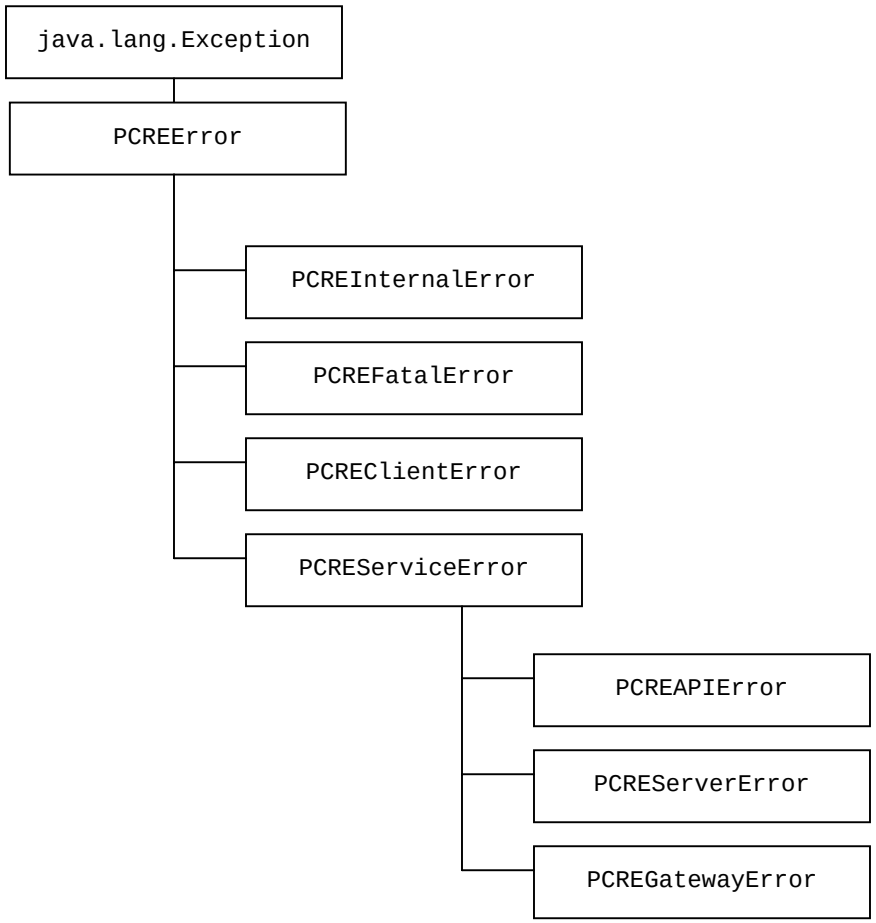


Figure 3.2

The six derived error classes are

- `PCREClientError` is used to identify errors that are detected by the Java client classes. These errors are caused by an invalid argument to a client-side method, or an inappropriate use of such a method.

- `PCREGatewayError` is used to identify errors that are detected in the Gateway Server and are reported back to the client. These errors can arise when the client tries to connect to the Gateway, or when the Gateway tries to start your Report Engine Server.

- `PCREServerError` is used to identify errors that are detected in the Report Engine Server but are not specifically related to the PCRE customization API. For example, they could be generated by a network failure.

- `PCREAPIError` is used to identify errors that are detected by the Crystal Reports Print Engine in the Report Engine Server. The Server transforms the error into an exception containing the Crystal error number and message, and then returns it to the client.

- `PCREFatalError` and `PCREInternal` error are used internally by the PCRE package to report catastrophic errors. An internal error usually indicates a logic failure within the code, while a fatal error indicates an external condition that makes it impossible to continue. When handling these errors, you should normally terminate your application in an appropriate way.

In addition to its constructors, each error class has a set of methods for throwing and reporting errors. The `Report` methods use a corresponding method in the `PCREConsole` class to display the error message. So if you call the `Report` method of `PCREAPIError`, then this method will in turn call the `PCREAPIErrorMsg` method of the client's console.

The hierarchy of error classes allows you to distinguish between various kinds of errors and to handle them individually or as a group. For example, in most of the programming examples we have used a simple try-catch block in the following way:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(“example.host.com”)
   client.Connect();
   PCREOpenEngine engine = client.Open();
   /* Open job, do API calls ... */
   engine.Close();
} catch ( PCREError e ) { e.Report(); }
```

Although this code is correct, errors with very different causes will transfer control to the same exception handler, and this may leave your program in an uncertain state. We can be more discriminating in the treatment of errors by using specialized try-catch blocks in a variety of ways.

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(“example.host.com”)
   /* Add client customization code here ... */
   client.Connect();
} catch ( PCREError e ) {                                    // H1.
   /* Handle all connection errors here. */
}

try {
   PCREEngine engine = client.OpenEngine();
   try {
      PCREJob job = engine.Open("C:\\ACME\\sales.rpt");

      /* job-processing: phase-1. */
      try {
         /* Customization method calls ... */
      } catch ( PCREAPIError e1 ) {                          // H2.
         /* Handle API errors in this phase here */
      }

      /* job-processing: phase-2. */
      try {
         /* Customization method calls ... */
      } catch ( PCREAPIError e2 ) {                          // H3.
         /* Handle API errors in this phase here */
      }

      job.Close();
   } catch ( PCREServiceError e3 ) {                         // H4.
      /* Handle server errors here */
   }
   engine.Close();
} catch ( PCREError e ) {                                    // H5.
   /* Handle all remaining errors here */
}

/* Disconnect client. */
client.Quit();
```

Handler H1 handles all possible connection errors. Since the actual errors thrown could be instances of the `PCREGatewayError`, `PCREServerError` or `PCREClientError` classes, the handler uses the `PCREError` base class to trap all three possibilities. You might want to evolve the handler so that a limited number of retries is allowed before terminating the application.

Handlers H2 and H3 handle `PCREAPIErrors` that occur within separate report customization phases. If any other server-side errors occur, they will be handled by the outer handler at H4. This arrangement allows us to proceed with confidence from phase 1 to phase 2. If an API error occurs in phase 1, then the H2 handler can decide whether to allow a retry, or to proceed on some alternative control path.

The handler at H5 catches all errors ignored by the inner handlers. In addition, if we fail to open a print job, we transfer to this point and subsequently call the client's `Quit` method to disconnect from the Report Server.

Table 3.8 shows how each error class's `Report` method calls a corresponding method of the `PCREConsole` interface.

Error Class	Console interface reporting method
`PCREClientError`	Calls `PCREClientErrorMessage`
`PCREGatewayError`	Calls `PCREGatewayErrorMsg`
`PCREServerError`	Calls `PCREServerErrorMsg`
`PCREAPIError`	Calls `PCREAPIErrorMsg`
`PCREFatalError`	Calls `PCREFatalErrorMsg`
`PCREInternalError`	Calls `PCREInternalErrorMsg`

Table 3.8

This association means that whenever you call the `Report` method on an error class (be it a base class like `PCREError`, or a derived class like `PCREAPIError`), the correct `PCREConsole` interface method for the actual error is always called. Of course you don't need to call the `Report` method at all. If you have trapped the error with an appropriate try-catch block, then how you handle it thereafter is entirely within your control.

In Java it is not possible to omit a try-catch block when calling a PCRE package method that throws an exception. If you forget, the Java compiler will report the inconsistency in your program.

Console Interface

The Parallel Crystal Java client package does not make any internal assumptions about the nature of your application. In particular, it does not know whether output messages will be sent to a character stream or displayed in a GUI. Consequently, the reporting facilities incorporated into the error handling classes output error messages using an abstract "application console" interface. You can adapt this interface to your application by providing an appropriate implementation class.

The Parallel Crystal Application Console interface is called `PCREConsole`. It contains seven methods that allow a message to be displayed according to the kind of information it conveys. For example, fatal error messages might be accompanied with a sound effect, whereas API errors might simply be written to a text window. You provide the appropriate implementation of these methods in your implementation class. If your application uses a simple character-based user interface, then the pre-packaged `PCREApplicationConsole` class may be adequate for your requirements.

You must install your application console by supplying it as the first argument of the `PCREApplicationClient` constructor. If you omit the argument, an instance of the character-based `PCREApplocationConsole` will be installed by default.

For example:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(“example.host.com”)
   client.Connect();
   ...
   try {
      /* API method calls ... */
   } catch ( PCREAPIError e1 ) { e1.Report(); }
   ...
} catch ( PCREError e2 ) { e2.Report(); }
client.Quit();
```

Will report all errors using the default `PCREApplicationConsole`.

If your application has a graphical user interface or your client is an applet, then the `PCREApplicationConsole` class is inappropriate, and you will require a graphical implementation of the `PCREConsole` interface.

A typical graphical console might take the following form:

```
public class GUIConsole implements PCREConsole {

   private void LogMsg( String msg ) {
      /* Log a message ... */ }

   private void MessageBox( int kind, String msg ) {
      /* Display a modal message box ... */
   }
   public void PCREOutputMsg( String msg ) {
      MessageBox(TEXT_MSG, msg);
   }

   public void PCREAPIErrorMsg( PCREError e ) {
      LogMsg(e.getText()):
   }

   public void PCREServerErrorMsg( PCREError e ) {
      MessageBox(SERVER_MSG, e.getText());
   }

   public void PCREGatewayErrorMsg( PCREError e ) {
      MessageBox(GATEWAY_MSG, e.getText());
   }

   public void PCREClientErrorMsg( PCREError e ) {
      MessageBox(CLIENT_MSG, e.getText());
   }

   public void PCREFatalErrorMsg( PCREError e ) {
      MessageBox(FATAL_MSG, e.getText());
   }

   public void PCREInternalErrorMsg( PCREError e ) {
      MessageBox(INTERNAL_MSG, e.getText());
   }

}
```

The console is installed in the client using a modified constructor call.

The code fragment above then takes the following form:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(
                  new GUIConsole("example.host.com"));
   client.Connect();
   ...
   try {
       /* API method calls ... */
   } catch ( PCREAPIError e1 ) { e1.Report(); }            // H1.
   ...
} catch ( PCREError e2 ) { e2.Report(); }                  // H2.
client.Quit();
```

The `PCREAPIError` caught in handler H1 will be reported with the `PCREAPIErrorMsg` method of the `GUIConsole`. However all errors caught by the handler H2 will also be reported with the correct `GUIConsole` method since `e2` is actually a reference to derived error class whose `Report` method overrides the base class method. So if `e2` happens to be a `PCREClientError`, it will be reported with the `PCREClientErrorMsg` method of the `GUIConsole`.

Parallel Crystal Report Retrieval

When a report is stored in a disk file, the file is generated on the Report Server machine using the path name you supplied in an `ExportTo` or `OutputTo` API call. If you wish to view the file, then you will normally have to retrieve it back to the client machine. The way in which you do this depends upon the report file format (PostScript, PDF, HTML or other), the nature of your Java client (application or applet), and the configuration of your Parallel Crystal installation. We'll explore all these possibilities in the following sections.

Retrieval to Application Clients

If you have an application client, you can use the Parallel Crystal Report Retrieval Service to return your report. You activate this service by calling the client's `setRetrieveMode` property method with the argument value `true`, and you specify a path for the report file on the *client's* machine rather than the Report Server machine. The following example shows how the Report Retrieval Service is used to return a PDF report to the client machine:

```
client.setRetrieveMode(true);                              // 1.
client.Connect();                                          // 2.

PCREEngine engine = client.OpenEngine();
PCREJob job =
   engine.OpenJob(                                         // 3.
       "C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");
job.OutputToPDF("C:\\MyReports\\Box.pdf", null);           // 4.
job.Start();                                               // 5.
job.Close();
engine.Close();
```

Each numbered paragraph below corresponds to the same numbered code statement

1. The automatic Report Retrieval Service is activated by calling `setRetrieveMode` with the value true.

2. The client is connected to the Report Server on `example.host.com`.

3. A print job is opened for `C:\MobileApps\pcre\SampleReports\Box.rpt`.

4. The report format is set to PDF and the path name `C:\MyReports\Box.pdf` specifies the output file on the client machine.

5. The print job is run. The report is generated on the Report Server machine and is retrieved to the file `C:\MyReports\Box.pdf` on the client machine.

If the report is generated correctly, then the file `C:\MyReports\Box.pdf` will be available on the client machine on return from the Start method at Step 6.

The Report Retrieval feature requires your Parallel Crystal to be installed with a file transfer service. If this service is not running then Step 5 above will throw a `PCREServerError` with the message:

> PCREServerError: Connect: cannot connect to PCRE File Transfer Service.

The Report Retrieval feature can be dynamically switched on or off for a client, by calling `setRetrieveMode` with the argument value `true` or `false`.

Retrieval to Applet Clients

If you have an applet client, then the easiest way to retrieve the file is to use the browser itself. However this restricts the output format of the report to HTML or PDF.

In Chapter 2 in the section entitled Using Parallel Crystal from a Java Applet, we described a ShowReport method that retrieves a report using a URL composed from the report file name and the Web Server host address. Naturally, the Web Server must be running on the same machine as the Report Server. If this is not the case, then it is possible to use the IIOP GateKeeper as a substitute Web Server. The GateKeeper is part of the Parallel Crystal installation so it is always available on the Report Server. However, it normally runs on external port 15000 so that you require a URL of the form http://www.mantis.com:15000/sales.html to retrieve the file from the host `www.mantis.com`. You should also note that the GateKeeper is not a high performance Web Server and can only serve HTML files.

The automatic Report Retrieval Service is not available for Java applet clients since it is generally not possible for applets to write to the file store.

Parallel Crystal Configuration Server

When Parallel Crystal is installed, a number of directories are created with sample Crystal Reports and demonstration materials. In order to help you locate path names for these directories, Parallel Crystal incorporates a Configuration Server that returns configuration- and installation-dependent information to the client.

Java clients access the Configuration Server through the methods of the `PCREConfigServer` service provider class that are summarized in Table 3.7 on page 73. Your client is automatically connected to the Configuration Server when you call the `Connect` method. Thereafter you can get a reference to the `PCREConfigServer` class by calling the client's `getConfigServer` method.

Method	**Purpose**
`GetPCREVersion`	Returns the Parallel Crystal version string
`getPCREHostName`	Returns the Parallel Crystal Report Server host name
`getPCREHostAddress`	Returns the Parallel Crystal Report Server host address
`getPCREInstallDir`	Returns the Parallel Crystal installation root directory
`getSampleReportDir`	Returns the Parallel Crystal sample reports directory
`getSampleOutputDir`	Returns the Parallel Crystal sample output directory
`getPCREWebServerDir`	Returns the Web Server's root directory
`getPCREGatewayName`	Returns the name of Parallel Crystal Gateway Server

Table 3.9

Here is a simple example that shows how to retrieve the host name of your Report Server and the version of Parallel Crystal that it is running:

```
client.Connect();
PCREConfigServer config = client.getConfigServer();
System.out.println("Connected to PCRE version " +
                   config.getPCREVersion() + " on " +
                   config.getPCREHostName());
```

The Configuration Server returns information that you can use with other service providers. The next example shows you how to connect to a Report Server and then set the Report Engine Server's working folder to the sample reports directory:

```
client.Connect();
String samples = client.getConfigServer().getSampleReportDir();
client.getServer().SetFolder(samples);
PCREEngine engine = client.OpenEngine();
PCREJob job = engine.OpenJob("Box.rpt");
...
job.Close();
engine.Close();
```

The report `Box.rpt` is part of the standard Parallel Crystal installation. However its full path name is likely to be installation-dependent. We therefore use the `getSampleReportDir` to return the path name of the sample reports directory and then call the `SetFolder` method of the `PCREServer` class to change the working folder of the Report Engine Server accordingly. Thereafter we can refer to all report files by their names relative to the sample directory.

Parallel Crystal Load Balancer

When your client has access to a group of Report Servers, you can use the Parallel Crystal Load Balancer service to ensure that the report processing load is distributed appropriately across the available server machines.

The load balancer provides two interfaces for

- Automatic use.
- Manual use.

Automatic Mode

When you use automatic mode, the client is connected to a Report Server chosen by the Load Balancer. The choice is made by a selection algorithm that is normally configured by the System Administrator.

The two algorithms most commonly used are

- Load factor - compares all the Report Servers in the group and returns the one with the smallest load.
- Round robin - simply cycles through all Report Servers in turn and selects the next one in the sequence.

Manual Mode

When you use manual mode, the Load Balancer returns a vector of Report Server hosts together with their load factors. You use the vector to make your own selection and then reset the host within the client. When you call the client's `Connect` method, you are connected to your chosen Report Server.

The Load Balancer is dynamically updated whenever a Gateway starts or stops a Report Engine Server, or when a new Gateway is started, or when an existing Gateway is stopped. This means that it adjusts to servers started by clients using the Load Balancing Service, and to servers started by clients who connected directly to a Report Server without using load balancing.

The following example shows you how to use the Load balancer in automatic mode:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(“LB.host.com”);       // 1.
   client.Connect(true);                                    // 2.
   /* make method calls ... */
} catch ( PCREError e ) { e.Report(); }                     // 3.
```

The connection process is completed in the following steps:

1. A `PCREApplicationClient` is specifying a host name for the Load Balancing.

2. `Connect` is called with the value `true` which activates the Load Balancer in automatic mode. The Load Balancer returns the Report Server host determined by its current selection algorithm and the client is connected to that host.

3. If there are no Report Servers available to the Load Balancer, the `Connect` call will throw a `PCREServerError` exception and at Step 3 the message

   Server error: Load Balancer failure: no available Report Servers

   will be printed.

In order to use the Load Balancer in manual mode, you call the `RequestReportServerList` method that returns a vector of candidate hosts. You apply your own selection procedure to the vector and then call the client's `ResetReportServerHost` method passing the selected vector element. Finally you call `Connect` to connect to the client to the Report Server. The following example shows a typical code fragment:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(“LB.host.com”)
   PCREReportServerInfo[] servers =
      client.RequestReportServerList();
   PCREReportServerInfo host = MySelection(servers);
   client.ResetReportServerHost(host);
   client.Connect();
   /* method calls ... */
} catch ( PCREError e.Report(); }
```

Each element of the vector returned by `RequestReportServerList` contains information about the Report Server encapsulated as an instance of the helper class `PCREReportServerInfo`:

```
public class PCREReportServerInfo {
   public double loadFactor;         // Gateway load factor.
   public int numberOfServers;       // Number of running servers.
   public int numberOfProcessors;    // Number of host processors.
   public String gatewayName;        // Gateway Server name.
   public String reportServerAddress;// Report server address.
}
```

The Report Server's load factor is given by the value of the `loadFactor` member and is currently equal to

```
NumberOfServers/NumberOfHostProcessors
```

The information given in the `PCREReportServerInfo` is subject to change, but is correct at the time of writing.

If you supply a host in the `PCREApplicationClient` (or `PCREAppletClient`) constructor call, then your client will use the Load Balancer service running on that host. However, the host settings for the Gateway and Configuration Servers are revised automatically whenever the client completes the connection to the selected Report Server.

In order to use the Load Balancer, your Parallel Crystal installation has to be configured to run this service. If the Load Balancer is not running then calls to `RequestReportServerList` and `Connect` will throw a `PCREServerError` exception with the message:

```
"Server error: cannot connect client to Load Balancer service"
```

Multi-threaded Clients

In Java, support for multi-threaded programming is provided in both the language and its platform-dependent implementation. The language provides simple synchronization primitives and their implementation in the Java Runtime Environment usually relies on the platform's native threads. Because threads are "built-in" to the Java language, you often have to be "thread-aware" even when building programs that you believe to be single-threaded. For example, if you build a GUI into your application, the underlying Abstract Windowing Toolkit will execute your event handlers on an internally allocated callback thread. Normally, this thread does not run concurrently with your program's "main" thread and so you do not have to worry about synchronization problems. However, very subtle problems can arise, when multiple client threads contend for AWT resources.

In order to support multi-threaded client applications, the PCRE Java classes provide instance-level thread-safety. This is achieved by making most property methods `synchronized` and by structuring the `PCREEngine` and `PCREJob` classes in such a way that their instance data is managed with a write-once, read-many access protocol. This allows multiple client threads to share instances of these classes without suffering undue synchronization overhead.

When you declare an instance of a `PCREApplicationClient` or PCREAppletClient, it holds a context that includes the thread, on which the constructor was invoked, the console for the client, the set of service provider classes, and the connection to the Report Server. Since each client has its own Report Engine Server, you can create applications in which multiple client threads connect to multiple Report Servers and generate reports concurrently. If your client is running on a multi-processor machine, then concurrency may be real rather than apparent.

The following example illustrates a multi-threaded client application in which 5 clients are concurrently connected to different Report Servers. Each client is created within a worker thread that executes independently of the others. Notice that the lifetime of the thread embraces the lifetime of the client that in turn embraces the lifetime of the Report Engine Server.

```
class ClientThread extends Thread {
   private String clHost;

   public ClientThread( String host ) {
      clHost = host;
   }

   public void run()
   {
      PCREApplicationClient client = null
      try {
         client = new PCREApplicationClient client(host);
         /* Connect client thread to host */
         client.Connect();
         PCREEngine engine = client.OpenEngine();
         /* do work ... */
         engine.Close();
      } catch ( PCREError e ) { e.Report(); }
      /* Disconnect client thread from host. */
      client.Quit();
   }

}

public class ThreadExample {
   private static final int NTHREADS    = 5;
   private static String[] hosts = { "beetle", "flea", "mantis",
      "butterfly", "grub" };

   public static void main( String[] args )
   {
      ClientThread[] clients = new ClientThread[NTHREADS];

      /* Create and start the client threads. */
      for ( int i = 0; i < NTHREADS; i++ ) {
         clients[i] = new ClientThread(hosts[i]);
         clients[i].start();
      }
      /* Wait for workers to terminate. */
      for ( int i = 0; i < NTHREADS; i++ ) {
         try {
            clients[i].join();
         } catch ( Exception e ) {
            System.out.println("Client " + i + ": " + e);
         }
      }
      System.out.println("All clients completed");
   }

}
```

It is especially important that each client thread calls the `Quit` method to disconnect from the Report Server. If you omit the call and let the thread die, then the connection to the Report Server will persist until the application as a whole terminates and the server process will remain on the server.

It is possible to adapt this example so that all client threads are connected to Report Servers via the Load Balancer. You simply omit the `host` parameter in the constructor call and call `Connect` with the argument value `true`. It is also possible to have the multiple client threads connect to the same host – possibly to produce different reports. In each case bear in mind that the actual performance of your client depends upon the number of processors you have available on your client's host. If there is only one, then your threads will execute concurrently but with interleaving in time, so there will be no performance gain.

Chapter 4
Trouble Shooting Java Clients

This chapter describes how to solve the problems that most frequently occur when developing Parallel Crystal clients in Java.

Problems with CLASSPATH

In order to construct a Parallel Crystal Java client you must import the PCRE package into your application with an import statement of the form

```
import com.mobileapps.pcre.*;
```

When the Java compiler encounters this statement, it uses an environment attribute called the CLASSPATH to locate the class files in the package. The CLASSPATH contains a list of directories separated by semicolons that tell the compiler where to start in its search for the package. If your Parallel Crystal Java Client distribution was installed on your client machine in the directory `C:\MobileApps\pcre`, then your CLASSPATH should have the form:

```
.;C:\MobileApps\pcre\jars\PCREjavaclient.jar
```

This tells the Java compiler to search for the PCRE package in the current directory, and then in the jar file `PCREjavaclient.jar`. Most compilers will also look in some implementation-dependent standard place before finally giving up the search. Unfortunately, as Java has evolved the way in which applications are distributed has changed and so has the way Java Development Environments use the CLASSPATH.

Currently, the Parallel Crystal Java Client Distribution is packaged as a single jar file called `PCREjavaclient.jar`. This file contains all the classes in the PCRE package and all the classes in the Inprise VisiBroker CORBA framework. When you install the Java Client, the installation procedure automatically modifies your CLASSPATH environment variable to include an entry for the `PCREjavaclient.jar` file.

Subsequently, the following problems can arise:

- Older versions of the Java compiler (such as that included with Microsoft Visual J++ Version 1.1) cannot read jar files. In this case you must unpack the jar file into a set of directories containing the individual class files. For example, if you use the command

  ```
  jar xvf C:\MobileApps\pcre\classes\PCREjavaclient.jar
  ```

  this will unpack all the class files into a directory hierarchy rooted in the `classes` sub-directory of `C:\MobileApps\pcre`.

You must then reset your CLASSPATH to

```
.;C:\MobileApps\pcre\classes
```

so that the compiler can locate the class files in their new location.

- Some Java Project Development Environments maintain their own classpath independently of the user's environment. If this is the case, then you need to ensure that you keep your development project classpath consistent with your environment CLASSPATH.

- You may be using the Parallel Crystal Java Client in conjunction with a Web Application Server. Many of the WebAS environments also use Java/CORBA products, and some use the Inprise VisiBroker product in common with Parallel Crystal. If both product installations modify your CLASSPATH you may need to check and reconcile the entries to ensure that your compiler will search the correct directories in the correct order.

- If you need to move your Parallel Crystal installation because of disk space management problems, it is best to uninstall and re-install the product, since this should restore and then re-modify your environment CLASSPATH accordingly. Of course, you may have to modify Project Development CLASSPATHs independently.

Common Problems with Client Connections

A call to your client's `Connect` method can fail for a number of reasons. Check these common situations first. If the problem persists, the causes may be more obscure.

The most common causes of connection failure are

- You specified a Report Server host in the client constructor call that is invalid: either the host name or the IP address is incorrect.

- You specified a correct host name or host address in the client constructor but the Report Server is down or is not running Parallel Crystal.

- Your client was able to connect to the Report Server but the Gateway could not start a Report Engine Server. Perhaps you specified incorrect command line arguments, or the machine was so busy that the Gateway timed out before the Server could get started. Or, perhaps the Parallel Crystal Installation has been moved and the Report Server's executable path name (which is configured at install time) is no longer valid. If you experience the timeout problem, use the `setWait` method of the `PCREGateway` service provider to increase the time in seconds.

For example:

```
PCREApplicationClient client = null;
try {
   client = new PCREApplicationClient(host)
   // Wait for up to 120 secs for connection.
   client.getGateway().setWait(120);
   client.Connect();
   /* API calls ... */
 } catch ( PCREError e ) { e.Report(); }
 client.Quit();
```

- You tried to connect using the Load Balancer but the Load Balancer could not locate any Report Servers. In this case it is possible that the Report Servers were not configured to use the Load Balancer Service, or there is an inconsistency in the way the Load Balancer Service has been configured. You should get your System Administrator to check the current Load Balancer configuration.

- The PCREGateway makes its initial reference available at a specific port. This port is determined in the following ways:

  - It first looks for an environment variable called IORPort that has been set to the desired port.

    Note: If the environment variable is present but its value can't be converted to a number, then it is ignored and the process continues to the next step.

  - Next, it looks for a registry entry called IORPort (see your System Administrator to setup or change this value), that is a DWORD registry value containing the desired port.

  - Finally, it defaults to 14020. When the PCREApplicationClient is instatiated, it can take an initial reference port as an argument. If a version without the port argument is used, then it uses the default of 14020.

  - If the client is trying to connect to a port that the server isn't using, then a connection failure will occur.

- The following connection problem can arise with badly configured Domain Name Servers. You may find that connections succeed when specifying a host with an IP address but fail when using a domain name. If this is the case, you should check with your System Administrator that the DNS servers on all machines are configured to handle both forward and reverse lookups.

Problems with Applets

First, consider this information about browsers

- If you are using Internet Explorer you need to explicitly enable it to run Java applets. To do this, select the Options Dialog with View|Options. Then select the Security tab and check the "Enable Java programs" box.

- If you are using Netscape Communicator output written to the System.out character stream is displayed in the Java Console. You can use this to display simple tracing messages. The Console is displayed by selecting Communicator|Java Console. If you select the console window and type an 'h' character, the console will display a set of single-character commands that you can use to dump memory and display threads.

- Netscape and Internet Explorer use different implementations of Java and this can cause problems with the appearance of your applet and programming errors that derive from the lack of support for the most recent Java features. You need to check Netscape and Microsoft web sites for the most recent information on their respective Java implementations.

The following checklist summarizes the points made in Chapter 2 in the section entitled Applet Client Connection:

- If Parallel Crystal is not running on the same machine as your Web Server, you need to run the IIOP GateKeeper as a "proxy server" on the Web Server. The GateKeeper forwards IIOP requests across the network to the Report Server and returns the replies. Check with your System Administrator that the GateKeeper is running.

- The GateKeeper can also be used as a simple Web Server. However it normally runs on port 15000 so if the Web Server host is www.acme.com, you should connect to the GateKeeper with a URL of the form

```
http://www.acme.com:15000/MyApplets/monthly.html
```

- Applet class files are normally downloaded from the same directory as the HTML page, and this directory is normally located relative to the Web Server root directory. Make sure you know what the root directory is and that your applet class file is indeed located where you think it should be.

- If you store applet class files in a different directory to the HTML files, then you need to specify a `codebase` attribute in the <applet> tag which specifies that directory relative to the Web Server root.

- To avoid version incompatibility problems, ensure that your applet downloads the CORBA framework included with the Parallel Crystal Java Client Distribution by including the following <param> tag between the <applet> and </applet> tags:

```
<param name=org.omg.CORBA.ORBClass
        value=com.visigenic.vbroker.orb.ORB>
```

- When connecting to the Report Server, your applet client needs to contact the IIOP GateKeeper. To do this, it needs to download a file called `gatekeeper.ior` from the Web Server. Normally the client expects to find this file in the same directory as the applet's class file. The Parallel Crystal installation normally includes this file in the directory `C:\MobileApps\pcre\ior`, so you can copy it to the required locations. The GateKeeper also generates the file each time it runs so you may find duplicate copies lying in other directories.

- If you don't want to copy the gatekeeper.ior file each time, you can include its location in a <param> tag included between the <applet> and </applet> tags:

```
<param name=ORBgatekeeperIOR
       value=http://www.acme.com/classes/gatekeeper.ior>
```

- If you know for sure that your Report Server and Web Server are on different machines then you will always need to use the GateKeeper as a "proxy server". You can save some connect time by including the following <param> tag between <applet> and </applet> tags:

```
<param name=ORBalwaysProxy value=true>
```

  This tag tells the client to contact the GateKeeper rather than try a direct connection to the Report Server that will fail.

- If your client lies behind a firewall, the GateKeeper should be configured to run with an accessible external IP address and port combination, and should also be enabled for HTTP tunneling if the firewall restricts internal traffic to the HTTP protocol. Your System Administrator should advise you on how best to deal with firewalls.

A number of other problems can occur which are characteristic of applet programming:

- Remember that browsers and web server have caches. Sometimes you can fix a problem in an applet but continue to get the old version of the class file because it was resident in a cache. When debugging, its best to re-start your browser each time, and if possible to re-start your web server.

- A browser won't download a class file if it finds it using the local CLASSPATH. This can be the cause of much confusion, so check your local file store for possible conflicts.

- Applets are an integral part of the Java Abstract Windowing Toolkit so their methods are invoked on callback threads. You should be aware that each call to the Report Engine Server (such as the `PCREJob Start` method) will block the thread for the duration of the call and freeze your GUI. You can make your GUI more responsive by making the Report Engine call on a background thread. This frees up the AWT callback thread allowing it to process other GUI events.

Problems with Export Formats

In Chapter 2 in the section entitled Print Job Outputs we described how to control the output format of the report with a variety of export formats.

A problem that can arise with report formats

- Not all of the formatting DLLs may be available on your Report Server. If you experience problems with formats other than HTML or PDF, then check with your System Administrator that the corresponding DLL is on your system and accessible to the Report Engine Server through the PATH environment variable.

Problems with PDF

This section describes the problems with PDF production using the Dynalivery library and Acrobat Distiller.

In Chapter 2 we described how to generate PDF reports using the `OutputToPDF`, `ExportTo` and `StartParallelPrintJob` methods of the `PCRJob` class and the `ParallelPrintReport` method of the `PCREEngine` class. The `OutputToPDF` method generates PDF using the Dynalivery PDF Library which is loaded directly into the Report Engine Server and which renders the report directly in PDF in a single step. The remaining methods produce PDF indirectly by generating a PostScript version of the report first of all, and then converting this to PDF using a utility from Adobe called Acrobat Distiller. The problems that arise when generating PDF therefore depend upon which of the methods you have called.

The following problems are known to occur when using the Dynalivery PDF Library with `OutputToPDF`

- You cannot currently generate "linearized" PDF for per-page retrieval to a browser. Future versions of the library may support this feature.

- OLE objects are rendered as bitmaps. Bitmaps don't display well when using the Acrobat zoom feature.

- Some graphics are implemented as bitmaps and cannot be magnified by the zoom-in feature in Acrobat Reader.

- Reports with overlapped images may also not render correctly. Future versions of the library may remove these limitations.

- Reports using fonts other than True Type fonts may have rendering problems.

In general, we recommend you try to generate PDF using the `OutputToPDF` method first of all. If problems arise, try the `ExportTo` method that generates PDF using Acrobat Distiller. If problems persist and you have access to Crystal Reports Designer, load your report back into the Designer and check it for errors.

The following problems are known to occur when using the Distiller with `ExportTo`

When the Report Engine Server receives a call to `ExportTo`, it saves the name of the PDF output file. Once the report has been generated in PostScript, the Report Engine starts the conversion to PDF by running a Parallel Crystal command line utility called `pdfdistiller`.

This program performs the following functions:

- It checks the input PostScript file exists.
- It checks that the output PDF file can be created.
- It runs Adobe Acrobat Distiller in an invisible desktop.
- It prevents multiple versions of Acrobat Distiller from running concurrently.
- It deletes the PostScript input file after conversion.

If you experience problems in generating PDF using `ExportTo`, check with your System Administrator that Acrobat Distiller has been installed correctly on your Report Server. The `pdfdistiller` utility searches the registry for the Acrobat Distiller's executable and expects to find an entry in

```
HKLM\Software\Microsoft\Windows\CurrentVersion\AppPaths\AcroDist.exe
```

If the entry no longer points to a valid file, `pdfdistiller` will not be able to execute the `AcroDist` program. If there is no registry entry, then `pdfdistiller` will attempt to run a program called `AcroDist` and will rely on the System PATH environment variable to locate this program.

You can run `pdfdistiller` manually on the Report Server to check its operation. The command line is

```
pdfdistiller [-command ] –input psfile –output pdffile
```

By default, the program runs silently and writes its error messages to the Windows/NT Event Log. You should check this log file before you do anything else. If you supply the optional `–command` argument then error messages will be written to the standard output stream as well.

Since Acrobat Distiller relies on the existence of the PostScript file, you should take care to ensure that the file does indeed exist before calling `pdfdistiller`. When you generate PDF via the `ExportTo` method, the Report Engine Server applies checks to ensure that the PostScript file has been completely generated before returning control to the server. This is because the PostScript file is not generated directly by the Server, but by the Windows Print SubSystem. This process can take several seconds on a busy machine processing large files.

Problems with the Report Server

The following problems can arise with the Report Server

- The Load Balancer and Report Retrieval Service can be turned on and off in the Report Server. If you get connection errors when trying to use either of these services, check with the System Administrator that the Report Server is configured to run them.

- If your client appears to block indefinitely inside a method of the `PCREEngine` or `PCREJob` class, then a third party component used by the Crystal Print Engine DLL may have created an error dialog that you are unable to see. To diagnose this problem, you need to get the System Administrator to run Parallel Crystal from the desktop so that all components are visible. If a Crystal API call leads to an error dialog, the Report Engine Server GUI will display the call, and the dialog itself will be visible. Parallel Crystal may be run in desktop mode using the *Start Parallel Crystal in Debug Mode* shortcut placed in the Start/Programs/Parallel Crystal menu during installation.

- Occasionally components of Parallel Crystal may crash or terminate abnormally. To assist error diagnosis, the Report Engine Server, File Server and Gateway Servers generate log files that contain trace and error messages. In addition, components write error messages to the Windows/NT event log. The log file may be viewed using the

- Occasionally your Java client may transfer to a catch-clause and report the error:

  ```
  "CORBA Error: CORBA UNKNOWN"
  ```

  The most likely cause is a Report Engine Server crash rendering the remote method that you are trying to call "unknown".

- Occasionally your Java client may transfer to a catch-clause and report the error:

  ```
  "CORBA Error: CORBA NOIMPLEMENT"
  ```

  This message means the method you are trying to call cannot be found in the Report Engine Server and the most likely cause is that the Parallel Crystal installations on your client and server machines are incompatible. You should get your System Administrator to make appropriate checks and re-install if necessary.

- Your Java client may report a `PCREAPIError` with the message:

  ```
  "PCRE API error: Logon failure"
  ```

  This is a standard Crystal Reports failure message. It occurs if the password and/or username requested by the data source are not supplied correctly. They are normally incorporated into the report file at design time, but you can override the entries when you call the `LogOnServer` method of `PCREEngine`. You should check that the details of the ODBC or other data source administrator on the Report Server match the settings on the machine where the report was designed. To help prevent this problem, we recommend that whenever possible, you design your reports in the same environment in which you expect to run them.

- A call to the `Start` method of `PCREJob` may report a `PCREAPIError` with the message:

  ```
  "PCRE API error: Request cancelled by user"
  ```

  This is a standard Crystal Reports failure message that can occur if the report cannot be sent to the nominated printer. You should check the availability of the printer or the print settings established by any calls to the `SelectPrinter` method of `PCREJob`.

Chapter 5
Getting Started with ActiveX Clients

This chapter describes how to write Parallel Crystal clients using Microsoft ActiveX/COM development tools such as Visual C++, Visual J++, Visual Basic and Active Server Pages.

The chapter is divided into six sections

- In the section entitled Parallel Crystal and ActiveX/COM clients, we explain the terms "ActiveX" and "COM" and describe how a related technology called "Automation" is used to enable compiled and interpreted Microsoft languages to access Parallel Crystal. Then we describe how automation-based clients control Report Customization through the methods and properties in the Parallel Crystal Report Engine Object Model.

- In the section entitled Using Parallel Crystal with Visual C++, we describe how to use the PCRE Object Model from C++ to produce simple reports in PostScript, PDF and HTML. First, we show how the low-level COM APIs are used, and then we show how the Visual C++ COM extensions provide a more effective interface for programming automation clients in C++.

- In the section entitled Using Parallel Crystal with Visual J++, we describe how to use the PCRE Object Model from Microsoft Visual J++ to produce reports in PostScript, PDF and HTML. We show how the Visual J++ Type Library Wizard generates wrapper classes that eliminate the need for detailed knowledge of COM, and make the programming of Java automation clients particularly easy.

- In the section entitled Using Parallel Crystal with Visual Basic, we describe how to use the PCRE Object Model from Visual Basic. Many aspects of COM Automation were designed specifically for the Visual Basic development community, and programming Visual Basic clients for Parallel Crystal is very simple. The examples in this section again show you how to produce reports in Postscript, PDF and HTML.

- In the section entitled Using Parallel Crystal with Active Server Pages, we describe how to use the PCRE Object Model from web server pages that include embedded scripts written in a simplified version of Visual Basic called VBScript. VBScript clients allow Report Customization to be integrated with Microsoft's Active Server Pages web development technology. You use the PCRE Object Model from VBScript in exactly the same way as you would use it from Visual Basic.

- The next section contains a rather lengthy introduction to COM. If you are anxious to get started, then skip directly to one of the following sections on using Visual C++, Visual J++, Visual Basic, or Active Server Pages.

Parallel Crystal and ActiveX/COM clients

This section describes how to access Parallel Crystal using Microsoft's COM/ActiveX technology. "ActiveX" is an umbrella term that was introduced in 1996 to embrace Microsoft's third generation OLE and COM technologies. Both OLE and COM are important because they allow all the major Microsoft development tools to access Parallel Crystal through a collection of COM objects provided by the Parallel Crystal Automation Server.

It is impossible to describe the Automation Server without some preliminary discussion of COM and COM Automation. COM itself describes how to build software from re-usable, object-based, components. Automation describes interfacing mechanisms that allow components to be controlled (or "automated") from high-level macro or scripting languages such as Visual Basic. The Parallel Crystal Automation Server provides a set of "automation objects" whose methods and properties allow COM enabled clients to access Parallel Crystal Report Servers. The object collection is hierarchical and reflects the logical structure of a Seagate Crystal Reports Document.

The following sub-sections provide preliminary material on COM, Automation, and the Automation Server. You'll find references to recommended textbooks for all the major Microsoft COM development tools later in the Chapter.

Microsoft COM

The Component Object Model specifies the requirements for binary re-usable object-based software components called COM objects.

These requirements include

- COM objects are specified by an interface that is a list of methods and properties. The properties embody the state of the object and the methods describe its functionality. Interfaces are expressed in the COM Interface Definition Language (IDL) that allows the methods, properties and parameters to be described in a C-like syntax. COM interfaces have names beginning with a capital I such as `ICRApplication`.

- COM objects enforce data encapsulation by exposing only their interfaces. Clients have no access to their data members, and no knowledge of the implementation details of the object's class.

- In order to use a COM object, clients must acquire a reference or pointer to its interface. Once acquired, the client is guaranteed that the object provides an implementation of all the methods and properties listed in the interface.

- COM clients neither create nor destroy COM objects directly. Clients cannot create COM objects because data encapsulation means that in general they do not know how much memory to allocate. Clients are also not permitted to destroy objects while they are in use.

- COM objects are implemented by co-classes that provide one or more COM interfaces. A co-class is created by an associated factory class called a "class-object" that is registered with the COM administration framework. COM co-classes have names beginning with a capital C such as `CRApplication`.

- COM objects monitor their clients using a simple system of reference counts. Whenever a client acquires a reference to the interface, the count is incremented, and whenever the client relinquishes the interface, it is obliged to call a `Release` method which decrements the count. The object will delete itself whenever its reference count reaches zero.

- Reference counts are maintained by a special interface called `IUnknown` that contains three methods called `AddRef`, `Release` and `QueryInterface`. Clients call `AddRef` and `Release` when they acquire and release interfaces, and use `QueryInterface` to make runtime queries for other interface pointers. For example, if a client has a reference to `ICat` that is provided by a co-class that also implements `IDog` and `IFish`, the client can call `QueryInterface` through `ICat` to get a reference to either `IDog` or `IFish`. This allows clients to discover new interfaces as they are added to existing objects.

- Every COM interface and co-class is identified by a universally unique identifier that is designed to prevent name clashes. However, each UUID has a symbolic representation in much the same way as an internet IP address has an a user-friendly domain name. When clients request an interface using `QueryInterface`, they use the symbolic form of the interface UUID. For example, the interface `IDog` is denoted by `IID_IDog`.

- Although COM interfaces are expressed in Microsoft COM IDL, the COM Specification is designed to allow COM classes and their clients to be programmed in a variety of conventional programming languages. However, COM specifies that clients must be able to call the methods of an interface through a runtime structure called a "vtable"[1] which is designed to eliminate language and linker implementation dependencies that impede binary re-usability in traditional programming environments.

- Software modules that implement COM classes are called "servers". Servers are generally referred to as "in-process", "local" or "remote", depending upon their location relative to the client. In general, clients are connected to the "nearest" server that provides the interface requested.

- COM allows interfaces to be attributed to an "apartment" which specifies how COM client and server threads may interact. The DCOM specification allows servers to specify security requirements that clients must satisfy before access to an interface is granted.

---

[1] A vtable is a vector of function pointers that it very similar to a C++ virtual function table.

When considering COM, it is important to realize that component programming is not the same as object-oriented programming. Components must be resilient, extensible, universally identifiable, and provide true "black box" re-usability. Simple classes implemented without addressing these requirements are unlikely to succeed as true software components.
COM embody an unfortunate reality that truly re-usable, encapsulated components are more complex to build and to use than dedicated C++ classes. As a result, Microsoft have added support tools to their Visual Programming Environments whose aim is to remove much of the "pain" of COM development. For Visual C++ these take the form of wrapper and template classes which are available as part of the Visual C++ COM SDK. For interpreted languages such as Visual Basic or Visual J++, COM support is provided through a related technology called Automation, and through extensions to the language interpreters that hide the low-level COM API.

We'll describe Automation in the next section. If you're curious to see how COM programming works from Visual Basic or Visual J++, take a look at the examples in the next section.

Microsoft COM Automation

Automation is an extension to COM that allows component objects to be accessed by weakly typed scripting languages. The term derives from "OLE Automation" which was originally designed to allow COM objects to be controlled by macros written in Visual Basic. The macros act as a kind of "glue" allowing object-based applications to be controlled by scripts performing complex tasks. Objects exposed in this way are called "automation objects" and their clients are called "automation controllers". To understand the need for automation, we need to compare the way in which compiled and interpreted clients access an object through its interface.

The methods and properties of a COM object are specified by an interface written in Microsoft Interface Definition Language (IDL).

For example, an abbreviated form of the `IUnknown` interface is:

```
interface IUnknown {
   HRESULT QueryInterface([in] REFIID riid,
                          [out, iid_is(riid)] void **ppvObject);
   ULONG AddRef();
   ULONG Release();
}
```

In order to use this with compiled clients, the interface is translated into an equivalent C++ or C header file. For example, the C++ version is as follows:

```
class IUnknown {
public:
   STDMETHOD(QueryInterface)( REFIId riid, void **ppvObject ) = 0;
   STDMETHOD_(ULONG, AddRef)() = 0;
   STDMETHOD_(ULONG, Release)() = 0;
}
```

This abstract class definition is packaged in a header file that is used by both the COM class providing the implementation of the three interface methods, and the clients that call these methods. This means that both the client and the implementation class work with the same (C++) definition of the interface that in turn means that the access code compiled for the client, will be consistent with the v-table compiled for the implementation. An interface that is implemented in this way, is called a "custom interface". Custom interfaces are very efficient because both the client and server sides can be compiled to code. However, the vtable layout is fixed at compile time and this constrains the interface to maintain a fixed set of methods with fixed parameter types. Clients that rely on statically typed interfaces are called "early-bound" clients.

It is clear that interpreted languages such as VBScript cannot access COM objects using a custom interface. Firstly, the interpreter cannot understand the C or C++ header files that are normally produced from the IDL interface definition. Secondly, the Microsoft IDL compiler that produces these header files cannot produce an equivalent VBScript header. Thirdly, the VBScript language has only one data type called a `VARIANT`, and does not understand even the simplest basic C types such as `int` and `char`. Finally, VBScript programs are not compiled in any conventional sense. They are interpreted on demand when the web page in which they are embedded is loaded into the Microsoft IIS Web Server.

The solution to these problems (called "automation") allows interpreted client languages to dynamically compose method calls at runtime using information about the interface recorded in a supplementary type library. Interfaces that can be used in this way are called "disp-interfaces" since they rely on a runtime dispatch mechanism to administer the call. The methods of a disp-interface are represented symbolically by numbers called DISPIDs, and are constrained to use a single universal type called a `VARIANT` for their arguments and results. COM contains a special generic interface called `IDispatch` that dispatches calls through its `Invoke` method. An object that provides a disp-interface is called an automation object, and its clients are called automation controllers[1].

Automation controllers are called "late-bound" clients because the methods they access are determined at runtime by their DISPID, rather than at compile time by their identifiers. Late-bound clients are unfortunately very inefficient since every method call entails a translation of names into DISPIDs, followed by a call to `Invoke` that in turn calls the target method. VBScript uses this mechanism in all its method calls. Scripting languages that are partly compiled like Visual Basic and Visual J++, are able to use disp-interfaces in an early bound fashion by performing the name to DISPID look-up at compile time. However, the latest versions of Visual Basic5 and 6 incorporate a compiler which allows compiled VB clients to access custom interfaces through their v-tables.

If Parallel Crystal is to be accessible to automation controllers such as VB or VBScript, its functionality must first be exposed in some COM-compatible way. This is achieved through the Parallel Crystal Object Model that we describe in the next section.

[1] The terms "automation controller" and "automation client" are used interchangeably in this manual.

The Parallel Crystal Object Model

The Parallel Crystal Object Model exposes the PCRE customization interface through a hierarchy of automation objects shown in Figure 5.1 on page 125. The hierarchy is designed to reflect the internal organization of a Crystal Report as a collection of areas, sections, and fields, and is managed through interfaces that conform to Microsoft's conventions for document-centric automation-enabled applications. In particular:

- The hierarchy is rooted at an Application Object offering the interface `ICRApplication`. This interface allows clients to connect to the Report Server and acquire a Parallel Crystal Report Engine.

- Clients use the `OpenReport` method of the `ICRApplication` interface to acquire a reference to one or more Report Objects. Clients can then customize and run these reports through the methods and properties of the `ICRReport` interface.

- The `ICRReport` interface conveys the structural composition of a report through a generic automation interface called a collection. Collections are designed to be used with the Visual Basic `For Each` statement that allows clients to iterate through each member in turn. Collections are identified by the rounded boxes in Figure 5.1, and by pluralized interface names.

- Child objects are created with properties that allow their immediate parent, the associated report object, and the root application object to be located.

- All interfaces are derived from `IDispatch` so that they are accessible from automation clients such as VBScript that cannot use custom v-tables.

The Parallel Crystal Object Model is based closely on the Object Library distributed with Seagate Crystal 6 and 7.[1] However, the PCRE Object Model contains extra methods to provide Parallel Crystal functions relating to remote connection, report retrieval and load balancing, and also omits many methods in the Seagate Library that are concerned with Report Design.

The Parallel Crystal Object Model is implemented by a component of the Parallel Crystal Client Distribution called the Parallel Crystal Automation Server that is described in the next section.

[1] The terms "Object Model" and "Object Library" are equivalent.

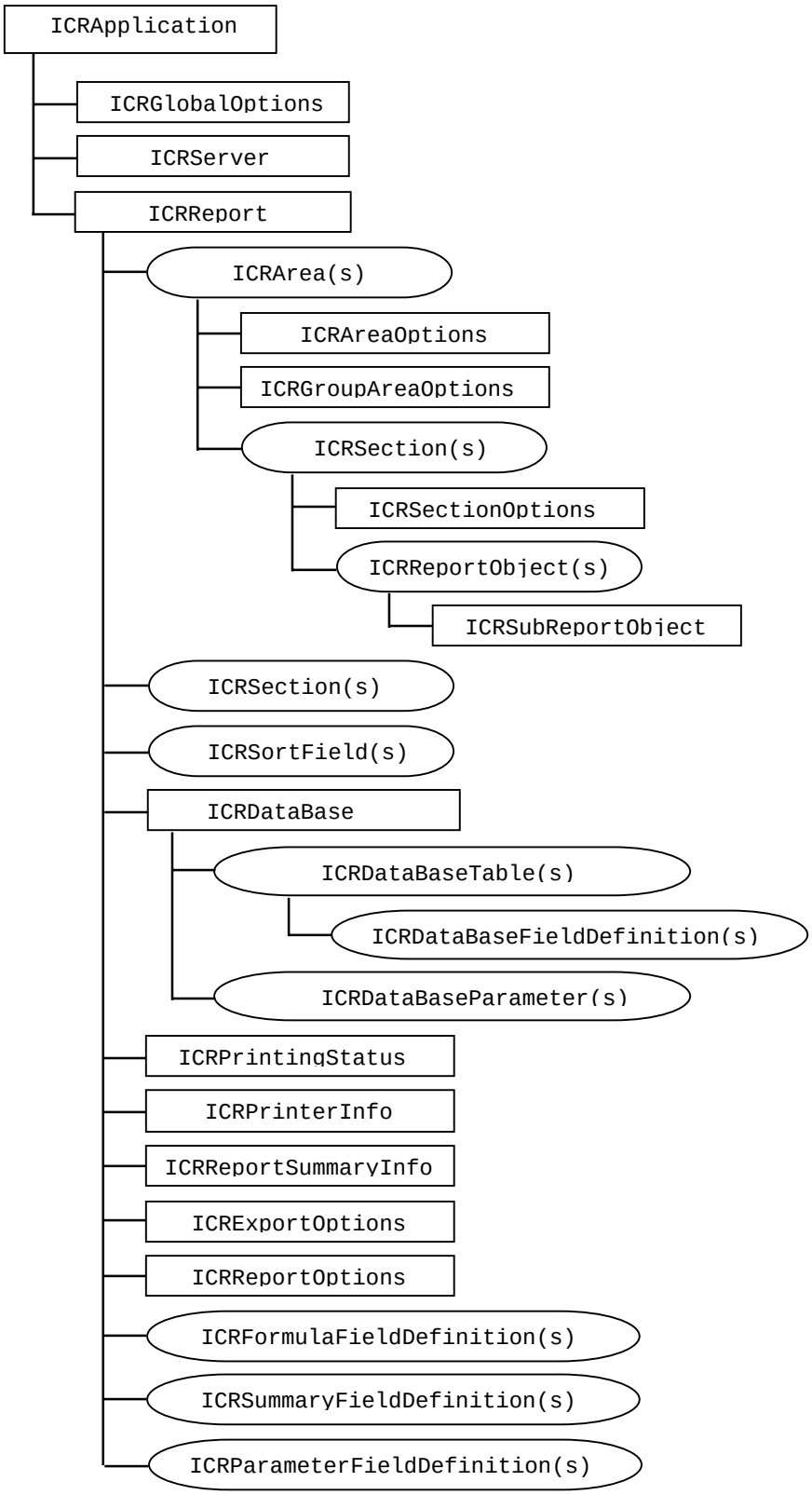

ICRApplication
ICRGlobalOptions
ICRServer
ICRReport
ICRArea(s)
ICRAreaOptions
ICRGroupAreaOptions
ICRSection(s)
ICRSectionOptions
ICRReportObject(s)
ICRSubReportObject
ICRSection(s)
ICRSortField(s)
ICRDataBase
ICRDataBaseTable(s)
ICRDataBaseFieldDefinition(s)
ICRDataBaseParameter(s)
ICRPrintingStatus
ICRPrinterInfo
ICRReportSummaryInfo
ICRExportOptions
ICRReportOptions
ICRFormulaFieldDefinition(s)
ICRSummaryFieldDefinition(s)
ICRParameterFieldDefinition(s)

Figure 5. 1

The Parallel Crystal Automation Server

The Parallel Crystal Automation Server is a DLL that provides the hierarchy of Automation Objects in the PCRE Object Model. Although it is referred to as a "server", the DLL must reside on the same machine as your client automation controller. If the controller is VBScript embedded in an ASP page, then the DLL should be installed on the Web Server's host. Details of the installation are given in the section Connecting to the Automation Server in the next Chapter.

When the Automation Server DLL is loaded, it uses the Parallel Crystal C/C++ client to establish a connection to the Report Server. In turn, this allows you to connect your controller to a Report Server on the same machine, or to a different (possibly) remote machine. A remote connection allows you to off-load the report-processing load, but necessarily introduces a small overhead in making customization API calls across a network. If your controller is VBScript running inside the IIS Web Server, this may be an acceptable price to pay to preserve the web server's performance.

The interfaces provided by the PCRE Automation Server are described in the online documentation that is normally located in the directory

```
C:\MobileApps\pcre\docs\AutomationServer\HTMLReference
```

If you inspect the Automation Server's type library with the OLE/COM Object Viewer that is available with Visual C++, then you may notice that the interfaces are described with IDL declarations such as the following:

```
[odl,
 uuid(AC73A0C1-BCDD-11D1-8C6B-000000000000),
 helpstring("ICRApplication Interface"),
 dual,
 oleautomation
]
interface ICRApplication: IDispatch {
   /* method declarations ... */
}
```

The attribute "dual" is important since it allows an interface to be accessed either as a disp-interface or as a custom interface. This means that the PCRE Automation Server can support automation controllers that can use efficient v-tables (such as VC++ and later versions of VB), as well as controllers such as VBScript that must use the slower disp-interface. In general, the Automation Server is easiest to use from VB dialects, almost as easy from Visual J++, and surprisingly hard from Visual C++! We'll examine each language interface in the following sections.

Using Parallel Crystal with Visual C++

This section describes how to write Visual C++ clients that use the PCRE Automation Server to generate simple reports in PostScript, PDF and HTML. In Generating PostScript Reports with COM, we show you how to use the low-level COM API to access the Automation Server. In the next section entitled Generating PostScript Reports with VC++ and COM, we show how Visual C++ COM support classes make it easier to develop automation clients in C++.

In order to understand the material in this section, you must be familiar with the COM function `CoCreateInstance`, and the `AddRef`, `Release` and `QueryInterface` methods of `IUnknown`. If you don't know COM at this level of detail, we suggest that you consider a simpler language such as Visual J++ or Visual Basic where much of this complexity is hidden. If you really want to learn COM using C++, we recommend first, that you read a good introductory text such as *Essential COM* by Don Box[1].

Generating PostScript Reports with COM

When you use the PCRE Object Model, you must first connect your client to the Automation Server, then use the `ICRApplication` interface on the root Application Object to connect to the Report Server. Once you have a Report Server connection, you use the `ICRApplication::OpenReport` method to return an `ICRReport` interface, and then use its methods to generate your report. As you acquire interfaces, the Automation Server generates a tree of object instances. You must remember to release the interfaces whenever you are finished with them, so that the tree can be systematically dismantled.

[1] Don Box: Essential COM, Addison-Wesley, 1998. ISBN 0-201-63446-5.

Example 5.1 illustrates the essential steps to produce a PostScript report. Certain details have been omitted for a first reading.

```
#include <pcreauto.h>                                          // 1.

int main( int argc, char *argv[] )
{
   HRESULT hr;                                                 // 2.
   ICRApplication *pApp = NULL;                                // 3.
   hr = CoInitialize(NULL);                                    // 4.
   if ( FAILED(hr) )
      return 0;

   try {
      hr = CoCreateInstance(CLSID_CRApplication, NULL,         // 5.
                            CLSCTX_INPROC_SERVER,
                            IID_ICRApplication,
                            reinterpret_cast<void **>(&pApp));
      if ( FAILED(hr) )                                        // 6.
         throw CRError("Cannot create PCRE Automation Server");

      // Connect to report server.
      CRStr host("flea.acme.com");                             // 7.
      hr = pApp->ConnectToReportServer(host);                  // 8.
      if ( FAILED(hr) ) throw CRError(pApp);

      // Open a report.
      IDispatch *pDispatch = NULL;
      ICRReport *pReport = NULL;
      CRStr name("C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");
      CRNoArg method;

      hr = pApp->OpenReport(name, method, &pDispatch);         // 9.
      if ( FAILED(hr) ) throw CRError(pApp);
      hr = pDispatch->QueryInterface(IID_ICRReport,            // 10.
                                     (void **)&pReport);
      if ( FAILED(hr) )
         throw CRError("ICRReport object not found");
      pDispatch->Release();                                    // 11.

      // Generate report.
      CRNoArg prompt, copies, collated, start, stop, file;
      hr = pReport->PrintOut(prompt, copies, collated,         // 12.
                             start, stop, file);
      if ( FAILED(hr) ) throw CRError(pApp);

      pReport->Release();                                      // 13.
      hr = pApp->DisconnectFromReportServer();                 // 14.
   } catch ( const CRError &error ) { error.Report(); }        // 15.

   if ( pApp != NULL ) pApp->Release();                        // 16.

   CoUninitialize();                                           // 17.
   return 0;
}
```

Example 5.1

Each numbered paragraph below corresponds to the same numbered code statement

1. In order to use the Object Model interfaces, their definitions must be included from a header file called `pcreauto.h`. This file is normally located in the installation directory

   ```
   C:\MobileApps\pcre\AutomationServer\include
   ```

   so you must ensure this directory is on your development environment's INCLUDE path.

2. All COM API calls return an `HRESULT` containing a 32-bit result code that is tested with the macros `SUCCEEDED` and `FAILED`. Consult a COM book for further details.

3. It is good COM programming practice to initialize variables that will receive interface pointers to NULL.

4. The COM API must be initialized with a call to `CoInitialize`. If the call succeeds, a matching call to `CoUninitialize` must be made before the application terminates.

5. The call to `CoCreateInstance` "connects" the client to the PCRE Automation Server. The call uses the CLSID supplied as the first parameter to locate the path name of the `cpemas1.dll` in the registry. The DLL is then loaded into your process so that interface method calls can be made. The call will fail if the DLL is not located on the client machine, or if the necessary information is not in the system registry. Otherwise, it will return a pointer to the `ICRApplication` interface.

6. The methods invoked through a COM interface are not allowed to throw exceptions. In order to simplify flow control, the client tests the `HRESULT` and throws a local exception for those that fail. The exception is an instance of a helper class called `CRError`.

7. COM API calls take pointers to wide character strings, rather than C-style character strings. In addition, disp-interfaces used by automation objects use a special kind of wide character string called a `BSTR` that is identical to Visual Basic's `String` data type. Raw `BSTR`'s are rather tedious to program, and so the example uses a `CRStr` helper class to manage string storage and the translations between normal and wide character formats.

8. The call to `ConnectReportServer` connects the client to a Report Server on the machine identified by the host name parameter. If the connection cannot be made, the call will return a failure `HRESULT`. The `ICRApplication` interface contains a `get_LastErrorString` method that is the reason for failure.

9. The call to `OpenReport` opens a print job for the report file and returns a pointer to the `IDispatch` interface on the Report Object.

10. Making C++ method calls through `IDispatch` is extremely tedious and very inefficient, so `QueryInterface` is called to return a pointer to the `ICRReport` custom interface. Methods on this interface are accessed using the v-table to make normal virtual function calls. The call to `QueryInterface` increases the reference count in the Report Object to 2.
11. There are no further accesses to the `IDispatch` interface, so it is discarded with a call to `Release` and the reference count on the Report Object falls to 1.

12. The report is generated by calling `ICRReport::PrintOut` to run the print job. This method takes six optional arguments that are initialized with values of type `CRNoArg`. Optional arguments were originally designed for Visual Basic or VBScript controllers and are not easy to use in C++. The `CRNoArg` provides a default value that allows the `PrintOut` method to ignore an argument that is "missing".

13. After the report is generated, the `ICRReport` interface is released. At this point the reference count on the Report Object falls to zero, and the object is deleted causing the corresponding print job to be closed. If this step is omitted, the Report Object will continue to exist until the application terminates.

14. The client disconnects from the Report Server with the call to `DisconnectFromReportServer`. If we wished, we could re-connect to a Report Server on another host with a subsequent call to `ConnectReportServer`.

15. If any method calls return a failure `HRESULT`, control transfers to this handler with an instance of the `CRError` which packages details of the error. The call to the `Report` method is intended to display the error in an appropriate way.

16. The `ICRApplication` interface is released causing the reference count on the Application Object to fall to zero prior to deletion. At this point, the client has relinquished all resources held on the Report Server and in the Automation Server.

17. The call `CoUnitialize` matches the `CoInitialize` call made at Step 4.

The nature of COM's error reporting and use of `BSTR` and `VARIANT` arguments make C++ automation controllers difficult to program. In the example, we used some simple wrapper classes to hide some of the implementation detail. We'll take a brief look at these wrappers in the next section.

1. Handling COM BSTR types

   In the example, we used a simple CRStr class to package BSTR and handle the conversions between wide and narrow characters. Here is a simplified version of this class:

```
class CRStr {
public:
   // Constructors and destructor.
   CRStr() : m_bstr(NULL), m_len(0), m_cstr(NULL) { }
   CRStr( const BSTR str ) : m_len(SysStringLen(str)) {
      m_bstr = SysAllocStringLen(str, m_len);
   }
   CRStr( const char *str ) : m_len(strlen(str)) {
      m_bstr = SysAllocStringLen(0, m_len);
      mbstowcs(m_bstr, str, m_len + 1);
   }
   CRStr( const CRStr& str ) : m_len(SysStringLen(str)) {
      m_bstr = SysAllocStringLen(str, m_len);
   }
   ~CRStr() {
      if ( m_bstr != NULL ) SysFreeString(m_bstr);
      if ( m_cstr != NULL ) delete[] m_cstr;
   }
   // Return the length of the string.
   UINT len() const        { return m_len; }
   // Conversion operator to BSTR.
   operator BSTR() const   { return m_bstr; }
   // Conversion operator to char *.
   operator char *() const {
      if ( m_cstr == NULL ) {
         m_cstr = new char[m_len + 1];
         wcstombs(m_cstr, m_bstr, m_len + 1);
      }
      return m_cstr;
   }
private:
   LPCSTR m_cstr;
   BSTR m_bstr;
   UINT m_len;
};
```

   The class uses the COM APIs SysAllocString and SysFreeString for BSTR allocation and de-allocation, and the mbstoscs and wcstombs Win32 APIs for string conversion. The destructor takes care of de-allocation automatically, which is particularly convenient in applications where exceptions will be thrown. However, you have to be careful. In the example, the strings host and name are not de-allocated until the try-block is exited at Step 15.

2. Handling COM Error Results

In order to re-package return codes as exceptions, we followed each method call with a statement like

```
if ( FAILED(hr) ) throw CRError(...);
```

where the `CRError` class is used to capture the details of the error. This class is defined as follows:

```
class CRError {
public:
   CRError( const char *msg ) : m_msg(msg) { }
   CRError( ICRApplication *pApp ) {
      BSTR str;
      HRESULT hr = pApp->get_LastErrorString(&str);
      m_msg = str; SysFreeString(str);
   }
   CRError( ICRReport *pRpt ) {
      BSTR str;
      HRESULT hr = pRpt->get_LastErrorString(&str);
      m_msg = str; SysFreeString(str);
   }
   CRError( const CRError& msg ) : m_msg(msg) { }

   operator CRStr() const { return m_msg; }

   ostream& Report() const {
      return cout << (char *)m_msg << endl;
   }
private:
   CRStr m_msg;
};
```

When the constructor is called with a string argument, the string is copied and stored in the `m_msg` member. However, when the constructor is called with an `ICRApplication` or `ICRReport` interface pointer, get`_LastErrorString` retrieves the error message generated by the last method call on the interface. In most cases, this string will have been returned from the Report Server.

3. Handling Optional `VARIANT` arguments

   We used a third wrapper class called `CRNoArg` that provides "missing argument" values for optional method arguments. The `CRNoArg` class does this by constructing a special kind of `VARIANT` value:

```
struct CRNoArg: public VARIANT {
   CRNoArg() {
      VariantInit(this);
      V_VT(this) = VT_ERROR;
      V_ERROR(this) = DISP_E_PARAMNOTFOUND;
   }
   ~CRNoArg() {
      VariantClear(this);
   }
};
```

   `PARAMNOTFOUND` values are generated internally by Visual Basic controllers, but must be passed explicitly when you use C++. The destructor clears the variant tag and de-allocates any storage. In the example, the variants initialized at steps 9 and 12 are not released until the try-block is exited at Step 15.

The helper classes used in the example are not provided as part of Parallel Crystal – largely because Visual C++ now provides sophisticated COM support through extensions to the C++ compiler and through the ATL template library. We'll show you how you can use these features in the section entitled Generating PostScript Reports with VC++ and COM.

Generating PDF and HTML Reports with COM

Fortunately, only trivial modifications are required to the example to generate reports in different output formats such as PDF and HTML.

- To generate a report in PDF and save the results, replace the call to `PrintOut` with the following call to `OutputToPDF`:

```
/* Step 12. */
CRStr file("C:\\MobileApps\\pcre\\SampleOutputs\\Box.pdf");
hr = report->OutputToPDF(file);
if ( FAILED(hr) ) throw CRError(report);
```

  When the method returns, the PDF report will be stored in the file `Box.pdf` in the `SampleOutputs` directory.

- To generate a report in HTML and save the results, replace the call to `PrintOut` with the following call to `ExportToHTML`:

```
/* Step 12. */
CRStr file("C:\\MobileApps\\pcre\\SampleOutputs\\Box.html");
hr = report->ExportToHTML(file);
if ( FAILED(hr) ) throw CRError(report);
```

  When the method returns, the HTML report will be stored in the file `Box.html` in the `SampleOutputs` directory. If the report has embedded images, the corresponding JPEG files will be stored in the same directory.

  The `ICRReport` interface has a number of other methods that you can call in the same way to generate reports in Excel or Microsoft Word format. There is also an `ExportToPDF` method that generates PDF using a PostScript to PDF conversion utility called Adobe Acrobat Distiller.

- To generate a report in PostScript and save it in a file, replace the call to `PrintOut` with the following call to `OutputToPS`:

```
/* Step 12. */
CRStr file("C:\\MobileApps\\pcre\\SampleOutputs\\Box.ps");
hr = report->OutputToPS(file);
if ( FAILED(hr) ) throw CRError(report);
```

  The report will be generated in the file `Box.ps` in the `SampleOutputs` directory. Note that you cannot depend upon the file being completely generated when you return from the method call. This is because the file must be generated by the Windows Printer Spooler that does not run synchronously with the Report Engine Server.

Notice that all these methods cause the print job to be started. Therefore, you must complete all necessary customization in previous method calls.

Generating PostScript Reports with Visual C++ and COM

In the previous sections, we showed you how to call the PCRE Automation Server with the low-level COM APIs. The examples showed that COM programming can involve much repetitive "boilerplate" code, and we used some wrapper classes to try to conceal this overhead. The wrappers were designed to address the following problems:

- COM disp-interfaces use a Visual Basic string type called a `BSTR`. C++ automation controllers need classes that allocate and de-allocate storage for this type, and manage the conversions between C-style and VB-style string formats.

- The COM API is intended to be language-independent and it is not possible to throw exceptions from a method of a COM-class. However, COM does possess APIs that allow a caller to retrieve error information deposited by the method, and to reconstruct a locally thrown exception.

- The methods of disp-interfaces can specify arguments using a Visual Basic type called a `VARIANT`. C++ automation controllers need classes that manage values of this type and supply default values for "optional" arguments.

To alleviate these problems, Microsoft have made substantial improvements to their Visual C++ Development Environment. In particular:

- Versions 4, 5 and 6 of Visual C++ provide a collection COM support classes together with wizards that generate wrapper classes for COM interface definitions. The wrappers use smart pointers to eliminate the need to call the `IUnknown` methods.

- Versions 4, 5 and 6 of Visual C++ provide the Active Template Library (ATL) that implements a complete framework for programming COM servers and clients.

- The Microsoft Foundation Classes provide support for programming COM/ActiveX visual controls.

It is not possible to include information on all of these support facilities in this manual. In the examples that follow, we will show how the VC++ compiler extensions and wrapper classes can be used to simplify access to the Automation Server. If you are interested in ATL, we suggest you consult a good introductory text such as the book by Richard Grimes[1]. The book by Mike Blaszczak[2] is one of several that covers MFC/COM programming.

[1] Richard Grimes et al: Beginning ATL COM Programming, Wrox Press, 1998. ISBN 1-861000-11-1.

[2] Mike Blaszczak: Professional MFC with Visual C++ 5, Wrox Press, 1997. ISBN 1-861000-14-6.

Example 5.2 shows how the VC++ COM wrapper classes are used to simplify access to the Automation Server:

```
#import <cpemas1.tlb> no_namespace                          // 1.
#include <iostream.h>

static void RunReport( void );

int main( int argc, char *argv[] )
{
   HRESULT hr;
   hr = CoInitialize(NULL);
   if ( FAILED(hr) )
      return 0;
   RunReport();                                             // 2.
   CoUninitialize();
   return 0;
}

static void RunReport( void )
{
   ICRApplicationPtr pApp;
   try {
      pApp(__uuidof(CRApplication));                        // 3.
      pApp->ConnectToReportServer("flea.acme.com");         // 4.
      _bstr_t name("C:\\SampleReports\\Box.rpt");           // 5.
      ICRReport* report =                                   // 6.
         static_cast<ICRReport *>(pApp->OpenReport(name));
      report->PrintOut();                                   // 7.
      report->Release();                                    // 8.
      pApp->DisconnectFromReportServer();                   // 9.
   } catch ( const _com_error& ) {                          // 10.
      cout << pApp->GetLastErrorString() << endl;
   }
}                                                           // 11.
```

Example 5.2

Each numbered paragraph below corresponds to the same numbered code statement

1. The `#import` statement instructs the VC++ compiler to use the type library in file `cpemas1.tlb` to generate a wrapper class for each interface in the PCRE Object Model. The wrapper classes are saved in files called `cpemas1.tlh` and `cpemas1.tli` that are created in your project's `Debug` or `Release` directories. The `cpemas1.tlh` file contains the class definitions for the wrappers, and the `cpemas1.tli` contains the inline wrapper methods. In a stable environment, these files will normally be created once, when you import `cpemas1.tlb` for the first time.

   The wrapper classes perform four main functions:

   - They declare "smart pointer" types for every COM interface. These pointers automatically manage object lifetimes by making calls to `AddRef` and `Release` from overloaded assignment operators. The constructors also handle calls to `CoCreateInstance` to bind a pointer to the implementation provided by its co-class.

   - They allow properties to be treated as variable accesses. If the property identifier occurs as an l-value, the put-property is called with the l-value expression as argument. If the property identifier occurs as r-value, the get property is called.

   - They allow method calls to omit optional arguments, and internally supply the correct default values.

   - They retrieve COM error information and incorporate it into a locally thrown exception.

2. The Application Object is created and destroyed from the constructor and destructor of a smart pointer type called `ICRApplicationPtr`. The example is restructured to provide a function `RunReport` that creates the smart pointer on entry, and destroys it on return.

3. The declaration manufactures an instance of the smart pointer `pApp` and binds it to the `CRApplication` co-class. The declaration is equivalent to steps_5_and_6 of the previous example. The constructor call takes a single argument that uses the VC++ `__uuidof` keyword to retrieve the CLSID for the `CRApplication` co-class. The smart pointer gives access to the methods of the `ICRApplication` wrapper class generated by the import statement.

4. The call to `ConnectToReportServer` executes the following wrapper class method internally:

```
inline HRESULT ICRApplication::ConnectToReportServer(_bstr_t ip)
{
    HRESULT _hr = raw_ConnectToReportServer(ip);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, _uuidof(this));
    return _hr;
}
```

   The parameter is specified with the COM support type `_bstr_t that` wraps `BSTR` in an appropriate class providing allocation and conversion methods for `BSTR`. The caller supplies an ASCII string and the constructors for `_bstr_t` automatically perform the conversion to `BSTR`. The body of the method wraps a local `HRESULT` that is examined after the actual method (renamed `raw_ConnectToReportServer`) returns. If the call fails, COM error information is retrieved and thrown as a local exception by a call to the COM helper function `_com_issue_errorex`.

5. The declaration of `name` shows explicit use of the `_bstr_t` type to create a wrappered `BSTR` initialized with a copy of the ASCII string `"C:\\SampleReports\\Box.rpt"`.

6. The `ICRApplication::OpenReport` method is called via the `pApp` smart pointer and the result is stored in a result pointer with type `ICRReport*`. This statement is executed in two steps. First, `OpenReport` is called via the following code:

```
inline IDispatch * ICRApplication::OpenReport (
   _bstr_t ReportFileName, const _variant_t & OpenMethod )
{
    IDispatch * _result;
    HRESULT _hr =
      raw_OpenReport(ReportFileName, OpenMethod, &_result);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, _uuidof(this));
    return _result;
}
```

   The method calls the "real" `OpenReport` that creates the co-class implementing the `ICRReport` interface. However, `OpenReport` is designed to work with automation controllers that are restricted to using disp-interfaces, and so its result is a pointer to the `IDispatch` interface on the same object.

   Although we could use `IDispatch` directly in C++, it is far more efficient to convert back to the dual `ICRReport` interface because we can then use the v-table to make efficient method calls. This conversion is performed implicitly by the `static_cast`. Strictly speaking, this is not correct COM programming practice. We'll discuss the alternatives in the section entitled The Trouble with Smart Pointers.

7. The call to `PrintOut` uses the wrapper class to provide the default values for missing arguments. The wrapper's method calls the "real" `OpenReport` method as follows:

```
inline HRESULT ICRReport::PrintOut (
   const _variant_t& PromptUser,
   const _variant_t& NumberOfCopy,
   const _variant_t& Collated,
   const _variant_t& StartPageNumber,
   const _variant_t& StopPageNumber,
   const _variant_t& OutputFileName )
{
   HRESULT _hr =
    raw_PrintOut(PromptUser, NumberOfCopy, Collated,
                 StartPageNumber, StopPageNumber,
                 OutputFileName);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, _uuidof(this));
    return _hr;
}
```

   The wrapper class declares the `PrintOut` method with six optional arguments. If the caller omits an argument, the wrapper supplies a "missing parameter" value.

8. `The ICRReport` interface is discarded by calling its `Release` method. The reference count is reduced to zero that deletes the Report Object and in turn closes the associated print job in the Report Engine Server.

9. The `DisconnectFromReportServer` method is called as before.

10. Calls to the `_com_issue_errorex` function raise exceptions of type `_com_error` that are trapped at this handler. In this example we use the body of the catch-clause to extract text of the last recorded error message by calling `ICRApplication::GetLastErrorString`. This method returns a `BSTR` but the wrapper class automatically provides the conversion to a `char*`.

11. The `pApp` smart pointer destructor is executed here, causing the `Release` method to be called on the `ICRApplication` interface. The reference count is reduced to zero that causes the Application Object to be deleted.

Clearly, the wrapper classes generated by `#import` combine with the `_bstr_t`, `_variant_t`, and `_com_error` classes to provide a far more "user-friendly" COM programming environment for C++. Full documentation on these Visual C++ features is available in the MSDN, and the book *Beginning ATL COM Programming* by Richard_Grimes also contains some very useful examples.

Generating PDF and HTML Reports with VC++ and COM

Changing the example in the previous section to generate reports in alternative formats such as PDF or HTML is trivial. In each case, the call to `PrintOut` in Step 8 is replaced with one that generates the report in the appropriate format. In particular:

- To generate a report in PDF and save the results, replace the call to `PrintOut` with the following call to `OutputToPDF`:

  ```
  /* Step 8. */
  report->OutputToPDF("C:\\SampleOutputs\\Box.pdf");
  ```

  When the method returns, the PDF report will be stored in the file Box.pdf in the `SampleOutputs` directory.

- To generate a report in HTML and save the results, replace the call to `PrintOut` with the following call to `ExportToHTML`:

  ```
  /* Step 8. */
  report->ExportToHTML("C:\\SampleOutputs\\Box.html");
  ```

  When the method returns, the HTML report will be stored in the file `Box.html` in the `SampleOutputs` directory. If the report has embedded images, the corresponding JPEG files will be stored in the same directory.

- To generate a report in PostScript and save it in a file, replace the call to `PrintOut` with the following call to `OutputToPS`:

  ```
  /* Step 8. */
  report->OutputToPS("C:\\SampleOutputs\\Box.ps")
  ```

  The report will be generated in the file `Box.ps` in the `SampleOutputs` directory. Note that you cannot depend upon the file being completely generated when you return from the method call. This is because the file must be generated by the Windows Printer Spooler that does not run synchronously with the Report Engine Server.

Remember, that all these methods cause the print job to be started. Therefore, you must complete all necessary customization in previous method calls.

The Trouble with Smart Pointers

In the preceding example, we used a smart pointer type `ICRApplicationPtr` to access the `ICRApplication` interface, and a "raw" `ICRReport*` pointer to access the `ICRReport` interface. However, `cpemas1.tlh` header file contains the definition of a smart pointer called `ICRReportPtr`, so why didn't we use it? The answer is that smart pointers are often not smart enough!

In the case of `ICRReportPtr`, the following difficulties arise:

- Since `ICRApplication::OpenReport` returns an `IDispatch*` pointer, it is necessary to convert this to a smart `ICRReportPtr` pointer. This conversion is possible via an `ICRReportPtr` constructor, but it has the hidden side effect of increasing the reference count on the report object by 1. You must therefore call `Release` to return the count to its correct value. So the conversion to the smart pointer type must take the form:

```
ICRReportPtr report = pApp->OpenReport(name);      // refcount 2!
Report->Release();                                 // refcount 1!
Report->PrintOut();
```

- A smart pointer will always call `Release` when its destructor is executed. This means that either you have to create artificial scopes or call the destructor manually in order to release the `ICRReport` interface.

If you combine these two requirements, they imply that you have to restructure the example as follows, in order to use the `ICRReportPtr` type:

```
try {
   pApp->ConnectToReportServer("flea.acme.com");
   // Artificial scope to ensure implicit report->Release
   {
      _b_strt name("...");
      ICRReportPtr report = pApp->OpenReport(name); // refcount 2!
      report->Release();                            // refcount 1!
      report->PrintOut();
      // Implicit Release call by ~ICRReportPtr destructor.
   }
   pApp->DisconnectFromReportServer();
} catch ( const _com_error& ) { ... }
```

The example avoided the use of smart pointers by using a `static_cast` to convert the value returned by `OpenReport` from an `IDispatch*` pointer to an `ICRReport*` pointer. Since `IDispatch` is the only base class of `ICRReport`, this cast is benign and does not change the pointer value. While strictly illegal according to the COM Specification, it nevertheless achieves the conversion without hidden side effects that affect the report object's reference count.

The strictly COM-conformant way of converting from `IDispatch*` to `ICRReport*` is as follows:

```
HRESULT *hr;
ICRReport* pReport;
IDispatch *pDispatch = pApp->OpenReport(name);
hr = pDispatch->QueryInterface(IID_ICRReport, (void **)&pReport);
pDispatch->Release();
```

The final call to `Release` releases the `IDispatch` interface. The `ICRReportPtr` smart pointer constructor does not perform this call, and that is why it was necessary to insert it manually.

Handling Errors from COM

When an error occurs in the Automation Server, it stores an error message internally and returns a generic COM error called `E_FAIL`. When you access the Automation Server using "bare" COM, you must test the result of each method call with the macro `FAILED`. If the return code indicates an error, you can recover the Automation Server's error message by calling the get_`LastErrorString` property of the `ICRApplication` and `ICRReport` interfaces. Earlier, we showed a helper class called CRError whose constructors re-packaged the error message in an exception.

Unfortunately, when you use the Visual C++ COM support classes, the pre-packaged `_com_error` does not include the Automation Server's message and you have to recover this in a context dependent way yourself. For example, the RunReport procedure from Example 5.2 on page 136 could be re-written as follows:

```
static void ReportError( LPCSTR source, LPCSTR msg )
{
   cout << source << ": " << msg << endl;
}

static void RunReport( void )
{
   ICRApplicationPtr pApp(__uuidof(CRApplication));
   try {
      pApp->ConnectToReportServer("flea.acme.com");
      _bstr_t name("C:\\SampleReports\\Box.rpt");
      ICRReport* report =
         static_cast<ICRReport *>(pApp->OpenReport(name));
      try {
         report->PrintOut();
      } catch ( const _com_error& e ) {
         ReportError(e.Source, report->LastErrorString);
      }
      report->Release();
      pApp->DisconnectFromReportServer();
   } catch ( const _com_error& e ) {
      ReportError(e.Source, app->LastErrorString);
   }
}
```

The respective catch-clauses access the `LastErrorString` property of the `ICRApplication` or `ICRReport` method, and the message is reported by the procedure `ReportError`. Notice that the helper classes generated in the `tli.h` header file allow the `LastErrorString` property to be accessed as if it was a data member.

Trouble Shooting Simple C++ COM Clients

In this section, we'll look at the most common errors that occur when programming simple clients using C++ with the Automation Server. We'll assume that errors are reported using the `ReportError` from the previous section, or something similar that uses the `LastErrorString` from the Automation Server.

- The call to the Application Object method `ConnectToReportServer` may fail and report the error:

  ```
  "No connection to a Server Report Engine has been made"
  ```

  There are two possible reasons for this error: either the argument string does not specify a valid host name, or the Parallel Crystal Report Server was not running on the specified host. Check that the string you supplied is either a valid host domain name or IP address, and then check that Parallel Crystal is installed and running on that host. If the problem persists, check Chapter 7, Trouble Shooting ActiveX Clients.

- The call to the Application Object method `OpenReport` may fail and report the error:

  ```
  "Invalid file name"
  ```

  This error occurs when the report file specified by the argument string does not exist on the Report Server. Check the location of the file and then supply its full path name to `OpenReport`.

- The call to the Report Object method `ExportToHTML` may fail and report the error:

  ```
  "Disk full"
  ```

  This highly misleading message is actually returned by the Report Engine and means that it was unable to create the HTML file to hold the generated report. It does *not* mean your disk is full! Check the argument string for `ExportToHTML` specifies a valid drive and path name, and that access permissions on the Report Server file store allow the file to be created.

- The call to the Report Object method `PrintOut` may fail and report the error:

  ```
  "Cancelled by user"
  ```

  This message is returned by the Report Engine when it is unable to locate a suitable printer. This is either because the printer has become inaccessible because of network problems, or because the printer settings inserted into your report at design time, are no longer valid. You may need assistance from either your System Administrator, or Dynalivery Technical Support Staff to fix this problem.

Things to Remember

- Independent of whether you use "bare" COM, or COM support classes, ALWAYS call `DisconnectFromReportServer` after you have finished generating your report. It will release resources on the Report Server and recover memory in the Automation Server.

- If you use "bare" COM with C++, make sure you understand the COM specification rules for calling `AddRef` and `Release`. In particular, ALWAYS call `Release` when you are finished using any interface pointer returned by the Automation Server.

- If you use COM smart pointer classes such as `_com_t`, make sure you understand when they will call `AddRef` and `Release`. Remember that a call to `QueryInterface` does an implicit `AddRef`.

Using Parallel Crystal with Visual J++

This section describes how to write Visual J++ clients that use the PCRE Automation Server to generate simple reports in PostScript, PDF and HTML. Since there is no direct Java interface to COM/ActiveX, Visual J++[1] provides a Type Library Wizard that generates Java wrapper classes for the interfaces provided by the Automation Server. These classes hide the details of COM completely so that you need only understand how to use the methods of the `ICRApplication` and `ICRReport` interfaces in order to generate simple reports.

Visual J++ Preliminaries

In order to use the PCRE Automation Server, you must generate the Java wrappers and "import" them into your project as follows:

- First, check that PCRE Automation Server has been installed AND registered on the machine running Visual J++.
- Run Visual J++ and create or select your Java project.
- Select the Visual Studio menu entry `Tools|Java Type Library Wizard`. A list box appears, similar to the one shown in Figure 5.2 on page 146.
- Search for the entry `cpemas1 1.0 Type Library` and check the box to the left.
- Click on OK. This causes the Type Library Wizard to read the PCRE Automation Server Type Library (which is held within the `cpemas1` DLL), and to generate a wrapper class for each interface in the library. The wrappers are compiled and stored in the sub-directory `cpemas1` of the Visual J++ "trusted" library at:

  ```
  C:\WINNT4.0\Java\TrustLib
  ```

  You do not have access to the Java source code for the wrappers. However, the wizard generates a text file called

  ```
  C:\WINNT4.0\Java\TrustLib\cpemas1\summary.txt
  ```

  which holds a text description of the methods and their parameters.

[1] The contents of this section apply to Visual J++ Version 1.1 or later, with Visual Studio 5 or later.

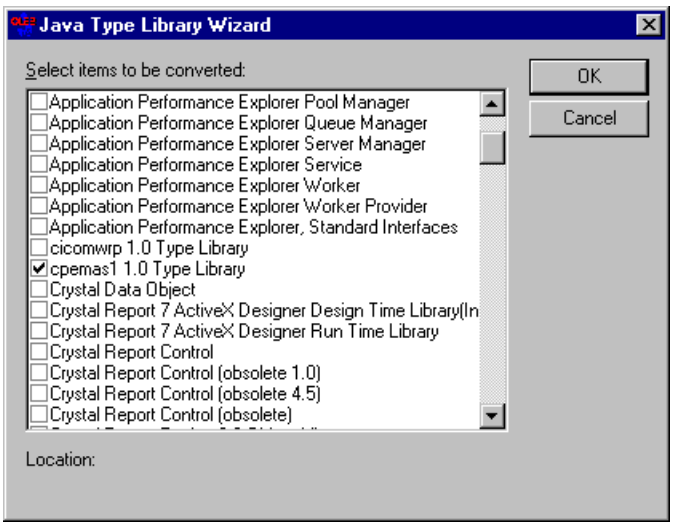

Java Type Library Wizard
Select items to be converted:
OK
Cancel
Application Performance Explorer Pool Manager
Application Performance Explorer Queue Manager
Application Performance Explorer Server Manager
Application Performance Explorer Service
Application Performance Explorer Worker
Application Performance Explorer Worker Provider
Application Performance Explorer, Standard Interfaces
cicomwrp 1.0 Type Library
cpemas1 1.0 Type Library
Crystal Data Object
Crystal Report 7 ActiveX Designer Design Time Library(In
Crystal Report 7 ActiveX Designer Run Time Library
Crystal Report Control
Crystal Report Control (obsolete 1.0)
Crystal Report Control (obsolete 4.5)
Crystal Report Control (obsolete)
Location:

Figure 5.2

Here is an extract from the file that shows how the root `ICRApplication` interface is expressed in Java:

```
public interface ICRApplication extends com.ms.com.IUnknown
{
    // Get properties.
    public abstract short getLastErrorCode();
    public abstract java.lang.String getLastErrorString();
    public abstract java.lang.String getserver();
    public abstract java.lang.Object getServerSettings();
    public abstract java.lang.Object getOptions();

    // Put properties.
    public abstract void putFTPUserName(java.lang.String);
    public abstract void putFTPPassword(java.lang.String);
    public abstract void putOSAgentAddress(java.lang.String);
    public abstract void putRetrieveMode(int);

    // Methods.
    public abstract void ClearError();
    public abstract void ConnectToReportServer(java.lang.String);
    public abstract void DisconnectFromReportServer();
    public abstract java.lang.Object OpenReport(java.lang.String,
       com.ms.com.Variant);
    public abstract void LogOffServer(java.lang.String,
       java.lang.String, com.ms.com.Variant,
       com.ms.com.Variant, com.ms.com.Variant);
    public abstract void LogOnServer(java.lang.String,
      java.lang.String, com.ms.com.Variant, com.ms.com.Variant,
      com.ms.com.Variant);
    public abstract boolean CanClose();
    public abstract void ParallelPrintReport(java.lang.String,
       int, int, java.lang.String, java.lang.String);
}
```

The extract shows that each method and property in the COM interface is mapped to a corresponding abstract method in the Java interface. In particular:

- `BSTRs` are mapped to Java Strings.
- `VARIANTS` are mapped to a Java class called `Variant` in the package `com.ms.com`.
- Java does not have optional arguments, so "missing" arguments have to be explicitly provided by the programmer.
- `IDispatch` pointers are mapped to Java Objects.
- The `ICRApplication` interface is derived from `com.ms.com.IUnknown`. However you do not have access to the `IUnknown` methods and so you cannot call `AddRef` or `Release` manually.

Generating PostScript Reports with Visual J++

Once you have generated the Automation Server Java interfaces, you can import them into your application with a normal import statement. The following example shows you how to generate a simple report in PostScript:

```
import cpemas1.*;                                          // 1.
import com.ms.com.*;                                       // 2.
class PostScriptExample {

   public static void main ( String[] args )
   {
      Variant none = new Variant();                        // 3.
      none.noParam();

      try {
         ICRApplication app =                              // 4.
            (ICRApplication)new CRApplication();
         app.ConnectToReportServer("example.host.com");    // 5.
         ICRReport report =                                // 6.
            (ICRReport)app.OpenReport("C:\\MobileApps\\pcre\\
                SampleReports\\Box.rpt", none);
         report.PrintOut(none, none, none, none, none, none);//7.
         app.DisconnectFromReportServer();                 // 8.
      } catch ( ComFailException e ) {                     // 9.
         System.out.println(e.getMessage());               // 10.
      }
   }
}
```

Example 5.3

Each numbered paragraph below corresponds to the same numbered code statement

1. The Automation Server interfaces are imported with a package import statement. The Visual J++ compiler includes the Java `TrustLib` directory on its search path so it should locate these classes automatically. If you have moved the `TrustLib` directory, then you may need to adjust your Visual Studio Option Settings accordingly.

2. The `com.ms.com` package contains additional COM support classes. A number of Automation Server methods require arguments to be supplied as values of a universal type called `VARIANT`, and this is done using methods of the `Variant` support class.

3. A number of methods in the Automation Server have optional arguments that are designed to work with Visual Basic. Unfortunately, you cannot mark a method parameter as optional in Java, and so you must supply the "missing" values by passing a `Variant` that has been initialized with the `noParam` method. In the example, we create a single missing value called `none`, and use it whenever the method expects an optional argument.

4. The root application interface is acquired by creating an instance of the `CRApplication` class and then casting to the `ICRApplication` interface. The wrapper classes force you to use this casting idiom and will throw an exception if you try to call the methods of the `CRApplication` class directly.

5. The client is connected to the Report Server on host `"example.host.com"`. This method will throw an exception if this host name is wrong, or the Report Server is not running, and control will transfer to Step 9.

6. The reference to the `ICRReport` interface is acquired by calling the `ICRApplication.OpenReport` method and specifying the name of the report file. The second parameter is optional and so we pass the "missing" value `none`. You must cast the result to the `ICRReport` interface type since you have no direct access to the Report Object's implementation class.

7. The report is generated and sent to the printer by calling the `ICRReport.PrintOut` method. This method takes six optional arguments and so six default "missing" values are supplied. The method returns when the report has been generated.

8. The call to `DisconnectFromReportServer` disconnects the client from the Report Server and releases resources at the client. You MUST call this method after your report has been generated. If you forget, resources will be tied up within your client and server machines until your application terminates.

9. The methods in the Java classes generated by the Type Library Wizard will raise exceptions if their corresponding COM methods fail. The exceptions can be trapped with a catch-clause that specifies a `COMFailException` type.

10. The error message supplied by the Automation Server can be retrieved with a call to the `getMessage` method, and the COM fail code can be retrieved with the `getHResult` method.

Unfortunately, the summary.txt file produced by the Type Library Wizard does not include the names of the formal parameters. You will need to consult the online documentation for the PCRE Automation Server to get a complete description of each method and its parameters.

Generating PDF and HTML Reports with Visual J++

The `ICRReport` interface provides several methods that make it easy for you to generate reports in other formats. In each case, the call to `PrintOut` in Step 7 in the preceding example is replaced with one that generates the report in the appropriate format. In particular:

- To generate a report in PDF and save the results, replace the call to `PrintOut` with the following call to `OutputToPDF`:

  ```
  /* Step 7. */
  report.OutputToPDF("C:\\SampleOutputs\\Box.pdf");
  ```

  When the method returns, the PDF report will be stored in the file Box.pdf in the `SampleOutputs` directory.

- To generate a report in HTML and save the results, replace the call to `PrintOut` with the following call to `ExportToHTML`:

  ```
  /* Step 7. */
  report.ExportToHTML("C:\\SampleOutputs\\Box.html");
  ```

  When the method returns, the HTML report will be stored in the file `Box.html` in the `SampleOutputs` directory. If the report has embedded images, the corresponding JPEG files will be stored in the same directory.

- To generate a report in PostScript and save it in a file, replace the call to `PrintOut` with the following call to `OutputToPS`:

  ```
  /* Step 7. */
  report.OutputToPS("C:\\SampleOutputs\\Box.ps")
  ```

  The report will be generated in the file `Box.ps` in the `SampleOutputs` directory. Note that you cannot depend upon the file being completely generated when you return from the method call. This is because the file must be generated by the Windows Printer Spooler that does not run synchronously with the Report Engine Server.

Remember, that all these methods cause the print job to be started. Therefore, you must complete all necessary customization in previous method calls.

Handling Errors from Visual J++

Although the generated wrapper classes allow you to use Java's exception handling idiom to trap errors, the messages returned by the `COMFailException.getMessage` method describe generic COM failures. For example, if a method in the Automation Server fails, it stores a failure message internally and returns a COM result called `E_FAIL`. When you print the message corresponding to this error, you get the rather terse and uninformative string

```
"Unspecified error"
```

In order to recover the message stored in the Automation Server, you must call the `getLastErrorString` method of either the `ICRApplication` or `ICRReport` interfaces. The following static method shows how a general error reporter can be written to take a reference to either interface, and recover the error message appropriately:

```
public static void ReportError( Object iref, ComFailException e )
{
   String msg = e.getMessage();
   if ( iref != null ) {
      if ( iref instanceof ICRApplication ) {
         ICRApplication appObject = (ICRApplication)iref;
         msg = appObject.getLastErrorString();
      } else if ( iref instanceof ICRReport ) {
         ICRReport rptObject = (ICRReport)iref;
         msg = rptObject.getLastErrorString();
      }
   }
   System.out.println("PCRE Error: " + msg);
}
```

Notice that we have to use Java's runtime-type operator `instanceof` to check whether the given `iref` is a reference to `ICRApplication` or `ICRReport`. Unfortunately, the types `ICRApplication` and `ICRReport` are not related by inheritance so we cannot use a shared definition of the `getLastErrorString method`. Neither can we use the COM `QueryInterface` method since it is hidden from us by the Visual J++ interface to COM.

In order to use `ReportError`, the PostScript example must be modified as follows:

```
public static void main ( String[] args )
{
   Variant none = new Variant();
   none.noParam();

   ICRApplication app = null;

   try {
      app = (ICRApplication)new CRApplication();
      app.ConnectToReportServer("example.host.com");

      ICRReport report =
         (ICRReport)app.OpenReport("C:\\MobileApps\\pcre\\
             SampleReports\\Box.rpt", none);

      try {
         report.PrintOut(none, none, none, none, none, none);
      } catch ( ComFailException e ) {
         ReportError(report, e);
      }

      app.DisconnectFromReportServer();
   } catch ( ComFailException e ) {
      ReportError(app, e);
   }
}
```

A similar effect can be achieved by writing an error handling class with two overloaded versions of `ReportError`, one of which takes an `ICRApplication` interface, and the other, an `ICRReport` interface.

Trouble Shooting Simple Visual J++ Clients

In this section we'll take a look at the most common errors that occur when programming simple clients using Visual J++. We'll assume that errors are reported using the `ReportError` from the previous section, or something similar that uses the `GetLastErrorString` from the Automation Server.

- The call to the Application Object method `ConnectToReportServer` may fail and report the error:

  ```
  "No connection to a Server Report Engine has been made"
  ```

  There are two possible reasons for this error: either the argument string does not specify a valid host name, or the Parallel Crystal Report Server was not running on the specified host. Check that the string you supplied is either a valid host domain name or IP address, and then check that Parallel Crystal is installed and running on that host. If the problem persists, check Chapter 7, Trouble Shooting ActiveX Clients.

- The call to the Application Object method `OpenReport` may fail and report the error:

  ```
  "Invalid file name"
  ```

  This error occurs when the report file specified by the argument string does not exist on the Report Server. Check the location of the file and then supply its full path name to `OpenReport`.

- The call to the Report Object method `ExportToHTML` may fail and report the error:

  ```
  "Disk full"
  ```

  This highly misleading message is actually returned by the Report Engine and means that it was unable to create the HTML file to hold the generated report. It does *not* mean your disk is full! Check the argument string for `ExportToHTML` specifies a valid drive and path name, and that access permissions on the Report Server file store allow the file to be created.

- The call to the Report Object method `PrintOut` may fail and report the error:

  ```
  "Cancelled by user"
  ```

  This message is returned by the Report Engine when it is unable to locate a suitable printer. This is either because the printer has become inaccessible because of network problems, or because the printer settings inserted into your report at design time, are no longer valid. You may need assistance from either your System Administrator, or Dynalivery Technical Support Staff to fix this problem.

Using Parallel Crystal with Visual Basic

This section describes how to write Visual Basic clients that use the PCRE Automation Server to generate simple reports in PostScript, PDF and HTML. The contents apply to 32-bit versions of Visual Basic 5 and 6.

Automation (which used to be called "OLE Automation"), was originally designed for Visual Basic 4, and many of its features (such as data access properties, `VARIANT`s and `BSTR`s), were borrowed from the Visual Basic language. The Visual Basic interpreter also hides the low-level details of COM so that you need only understand how to access the methods of the `ICRApplication` and `ICRReport` interfaces in order to generate simple reports. If you're curious to learn more about COM from a Visual Basic perspective, then we recommend the book by Ted Pattison[1].

Visual Basic Preliminaries

In order to access the PCRE Automation Server, you must import the Type Library into your Visual Basic Development Project using the Project References display shown in Figure 5.3 on page 154.

- First, check that PCRE Automation Server has been installed AND registered on the machine running Visual Basic.
- Start the Visual Basic Development Environment and create or select your VB project.
- Select the VB menu entry `Project|References`. The list box shown in Figure 5.2 appears.
- Locate the entry `cpemas1 1.0 Type Library` and check the box to the left.
- Click on OK.

[1] Ted Pattison: Programming Distributed Applications with COM and Microsoft Visual Basic 6.0.

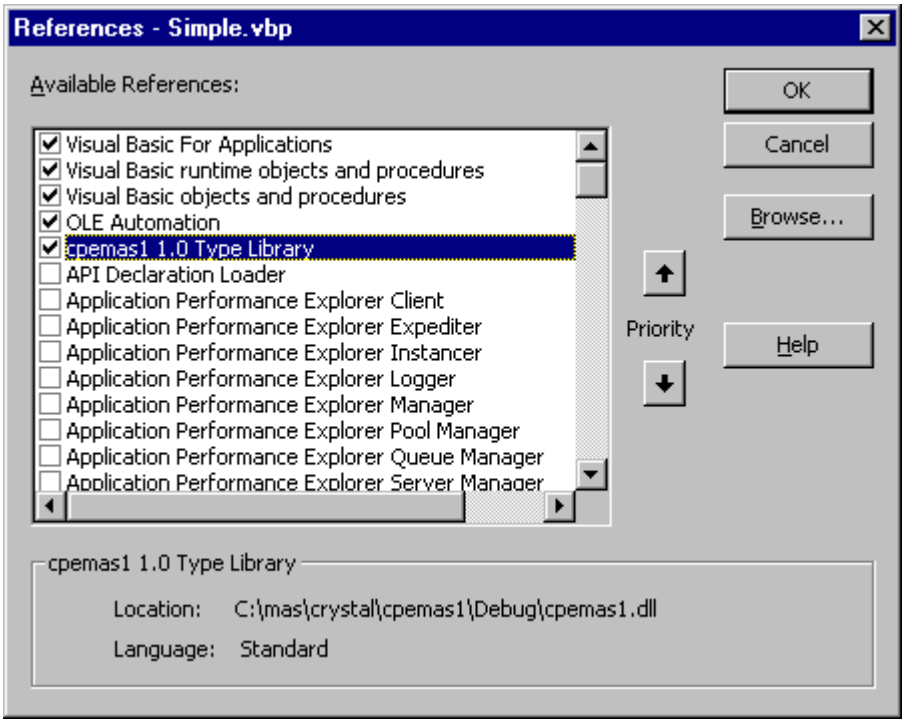


Figure 5.3

Once the PCRE Automation Server is imported into your project you can access the properties and methods of the root `ICRApplication` interface through Visual Basic `Dim` statements.

Generating PostScript Reports with Visual Basic

In this section we show how to construct a Visual Basic client that uses the Parallel Crystal Automation Server to generate PostScript reports. The client has a very simple user interface, which is shown in Figure 5.4 on page 155.

The name of the Report Server host and the path name of the report file are entered into the respective text boxes. When the Connect button is pressed, the client is connected to the Report Server and the Run and Disconnect buttons are enabled. When the Run button is pressed, the report is generated and a pop-up message box will confirm success. The Disconnect button disconnects the client from the Report Server. Failure to connect to the Report Server or to locate the report file, are indicated with appropriate error messages.

The components on the form are labeled as follows:

`lblHost`	Host label
`txtHost`	Host text box
`lblReport`	Report label
`txtReport`	Report text box
`cmdConnect`	Connect button
`cmdRun`	Run button
`cmdDisconnect`	Disconnect button

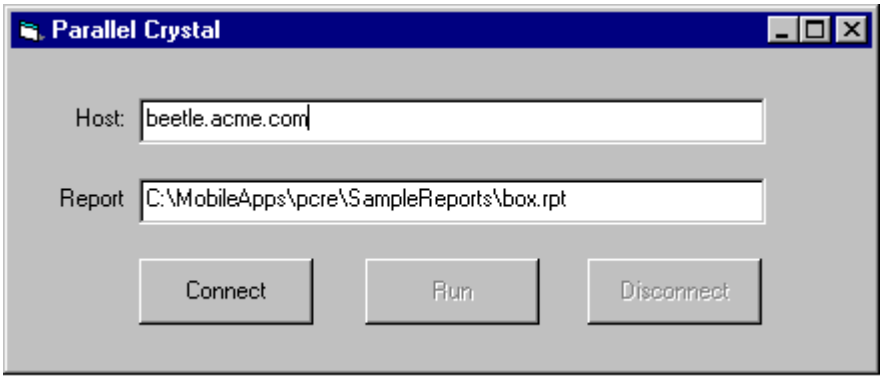


Figure 5.4

The code for the form contains the following outline sections:

```
Private app As ICRApplication                          ' 1.
Private bConnected As Boolean                          ' 2.

Private Sub setButtons( cb As Boolean, rb As Boolean, _  ' 3.
                        db As Boolean)
Private Sub Form_Load()                                ' 4.
Private Sub Form_Unload(Cancel As Integer)             ' 5.
Private Sub cmdConnect_Click()                         ' 6.
Private Sub cmdRun_Click()                             ' 7.
Private Sub cmdDisconnect_Click()                      ' 8.
```

Each numbered paragraph below corresponds to the same numbered code statement

1. The global variable `app` holds a reference to the `ICRApplication` interface on the PCRE Application Object. The details of the interface (such as its methods and properties) are recovered from the Automation Server Type Library.

2. The global variable `bConnected` is set to `True` when the client is connected to the Report Server, and `False` when it is disconnected.

3. The procedure `setButtons` is a utility that is used to set the `Enabled` property of each of the command buttons:

```
Private Sub setButtons( cb As Boolean, rb As Boolean, _
                        db As Boolean )
   cmdConnect.Enabled = cb
   cmdRun.Enabled = rb
   cmd.Disconnect = db
End Sub
```

4. The procedure `Form_Load` is called when the application form is loaded for the first time:

```
Private Sub Form_Load()
   On Error GoTo fail
   bConnected = False
   Set app = New CRApplication
   setButtons True, False, False
   Exit Sub

fail:
   MsgBox "Failed to connect to PCRE Automation Server"
   Set app = Nothing
End Sub
```

   The procedure creates a new instance of the PCRE Application Object and stores a reference to it in the variable `app`. If the call to `New` fails, the COM error is automatically detected by Visual Basic, and control will transfer to the error label `fail`. If the call to `New` is successful, the Connect button is enabled. Note that the argument to `New` is the name of `CRApplication` co-class that provides the `ICRApplication` interface.

5. The procedure `Form_Unload` is called when the application is terminated:

```
Private Sub Form_Unload( Cancel As Integer )
   If bConnected Then cmdDisconnect_Click
   Set app = Nothing
End Sub
```

   If the client is still connected to the Report Server, the `cmdDisconnect_Click` procedure is called to disconnect. Then the variable `app` is reset to `Nothing` that releases the single reference to the Application Object and causes it to be deleted.[1].

[1] Setting a VB object reference variable to Nothing is equivalent to calling Release on the associated interface.

6. The procedure `cmdConnect_Click` is executed when the Connect button is pressed:

```
Private Sub cmdConnect_Click()
   On Error GoTo fail
   If Len(txtHost.Text) = 0 Then
      MsgBox "Enter Report Server host name"
      Exit Sub
   End If
   app.ConnectToReportServer txtHost.Text
   setButtons False, True, True
   bConnected = True
   Exit Sub

fail:
   MsgBox app.LastErrorString
End Sub
```

   The procedure checks that a host name has been specified, and then calls the Application Object's `ConnectToReportServer` method. If the connection is established, the Run and Disconnect buttons are enabled, and the `bConnected` variable is set to `True`. If the client cannot connect to the Report Server, the `LastErrorString` method is called to return the error message.

7. The procedure `cmdRun_Click` is executed when the Run button is pressed:

```
Private Sub cmdRun_Click()
   On Error GoTo fail
   Dim report As ICRReport

   ' Check report file path name.
   If Len(txtReport.Text) = 0 Then
      MsgBox "Enter report file path name"
      Exit Sub
   End If

   ' Generate report.
   Set report = app.OpenReport(txtReport.Text)
   report.PrintOut
   MsgBox "Report generated"

   Exit Sub

fail:
   MsgBox app.GetLastErrorString
End Sub
```

   The procedure checks the report file path name has been specified, and then calls the Application Object's `OpenReport` method. This method opens a print job in the Report Engine Server and returns a reference to the `ICRReport` interface on the corresponding Report Object. The details of the interface (such as its methods and properties) are recovered from the Automation Server Type Library.

If the report is opened successfully, the method `PrintOut` is called to generate and print a PostScript version of the report. The method returns when the report has been generated.

When the control returns from the procedure, the `report` variable is implicitly reset to `Nothing`. This releases the interface causing the Automation Server to close the print job in the Report Engine Server and delete the Report Object.

Note that on return from this procedure the client is still connected to the Report Server. If the Run button is pressed again, a new report will be generated using the path name currently entered in the Report text box.

8. The procedure `cmdDisconnect_Click` is executed when the Disconnect button is pressed:

```
Private Sub cmdDisconnect_Click()
   On Error GoTo fail
   app.DisconnectFromReportServer
   setButtons True, False, False
   bConnected = False
   Exit Sub

fail:
   MsgBox app.GetLastErrorString
End Sub
```

The procedure disconnects the client from the Report Server and re-sets the command buttons. If the Connect button is pressed again, the client will be re-connected to the Report Server on the host currently entered in the Host text box.

Generating PDF and HTML Reports with Visual Basic

The `ICRReport` interface provides several methods that make it easy for you to generate reports in other formats. In each case, the call to `PrintOut` in procedure `cmdRun_Click` is replaced with one that generates the report in the appropriate format. In particular:

- To generate a report in PDF and save the results, replace the call to `PrintOut` with the following call to `OutputToPDF`:

```
report.OutputToPDF "C:\\SampleOutputs\\Box.pdf"
```

When the method returns, the PDF report will be stored in the file Box.pdf in the `SampleOutputs` directory.

- To generate a report in HTML and save the results, replace the call to `PrintOut` with the following call to `ExportToHTML`:

```
report.ExportToHTML "C:\\SampleOutputs\\Box.html";
```

When the method returns, the HTML report will be stored in the file `Box.html` in the `SampleOutputs` directory. If the report has embedded images, the corresponding JPEG files will be stored in the same directory.

- To generate a report in PostScript and save it in a file, replace the call to `PrintOut` with the following call to `OutputToPS`:

```
report.OutputToPS "C:\\SampleOutputs\\Box.ps"
```

  The report will be generated in the file `Box.ps` in the `SampleOutputs` directory. Note that you cannot depend upon the file being completely generated when you return from the method call. This is because the file must be generated by the Windows Printer Spooler that does not run synchronously with the Report Engine Server.

Remember that all these methods cause the print job to be started. Therefore, you must complete all necessary customization in previous method calls.

Handling Errors from Visual Basic

When an error occurs in the Automation Server, it stores an error message internally and returns a generic COM error called `E_FAIL`. When Visual Basic detects a COM error, it posts a message box containing the error code and the name of the Automation Server method that raised the error. For example, if the `ConnectToReportServer` method fails, Visual Basic will produce the message shown in Figure 5.5. The run-time error code is the value `E_FAIL` that is returned by `ConnectToReportServer` when it is unable to connect to the Report Server specified by the argument string.

In order to display error messages generated by the Automation Server, you must access the `LastErrorString` property of the `ICRApplication` and `ICRReport` methods. In the example, the `On Error Goto` statement transfers control to the labeled error handler whenever the Automation Server returns an error. You can choose at that point whether to report the error using the `LastErrorString`, or access the `Err.Description` property, or simply produce your own error message.

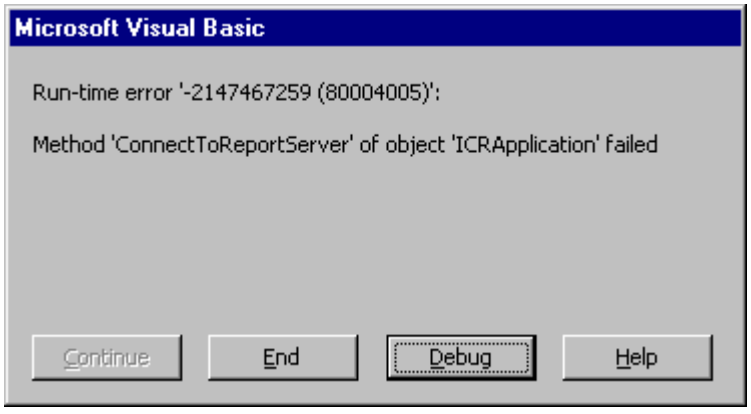


Figure 5.5

Unfortunately, the presence of `LastErrorString` properties in both the `ICRApplication` and `ICRReport` interfaces means that on reaching an error handler, you must know which one to call. For example, in `cmdRun_Click`, the handler calls `app.LastErrorString`. This is correct if the error occurred in `app.OpenReport`, but incorrect if the error occurred in `report.PrintOut`. There are various solutions to this problem:

1. You can restructure the code so that calls to `ICRReport` methods occur in a different procedure whose handler need only ever call `ICRReport.LastErrorString`.

2. You can set context flags so that a common handler can choose the correct `LastErrorString` to call.

3. You can use the Visual Basic `On Error Resume Next` statement that will return control to the statement following the Automation Server method call. Then you must test the result code in `Err` and report an error if required. For example:

```
On Error Resume Next
...
Set report = app.OpenReport(txtReport.txt)
If Err.Number <> 0 Then
   MsgBox "OpenReport: " & app.LastErrorString
   Exit Sub
End If

report.PrintOut
If Err.Number <> 0 Then
   MsgBox "PrintOut: " & report.LastErrorString
   Exit Sub
End If
```

Neither of these solutions is entirely satisfactory.

Trouble Shooting Simple Visual Basic Clients

In this section we'll take a look at the most common errors that occur when programming simple clients using Visual Basic. We'll assume that errors are reported by accessing the appropriate `GetLastErrorString` property in the Automation Server.

- The call to the Application Object method `ConnectToReportServer` may fail and report the error:

  ```
  "No connection to a Server Report Engine has been made"
  ```

  There are two possible reasons for this error: either the argument string does not specify a valid host name, or the Parallel Crystal Report Server was not running on the specified host. Check that the string you supplied is either a valid host domain name or IP address, and then check that Parallel Crystal is installed and running on that host. If the problem persists, check Chapter 7, Trouble Shooting ActiveX Clients.

- The call to the Application Object method `OpenReport` may fail and report the error:

  ```
  "Invalid file name"
  ```

  This error occurs when the report file specified by the argument string does not exist on the Report Server. Check the location of the file and then supply its full path name to `OpenReport`.

- The call to the Report Object method `ExportToHTML` may fail and report the error:

  ```
  "Disk full"
  ```

  This highly misleading message is actually returned by the Report Server and means that it was unable to create the HTML file to hold the generated report. It does *not* mean your disk is full! Check the argument string for `ExportToHTML` specifies a valid drive and path name, and that access permissions on the Report Server file store allow the file to be created.

- The call to the Report Object method `PrintOut` may fail and report the error:

  ```
  "Cancelled by user"
  ```

  This message is returned by the Report Server when it is unable to locate a suitable printer. This is either because the printer has become inaccessible because of network problems, or because the printer settings inserted into your report at design time, are no longer valid. You may need assistance from either your System Administrator, or Dynalivery Technical Support Staff to fix this problem.

Using Parallel Crystal with Active Server Pages

This section describes how to write an Active Server Pages client to generate simple reports in PostScript, PDF and HTML. The contents apply to ASP 2.0 running under Microsoft's IIS 4.0 Web Server.

Active Server Pages is a web development technology that allows HTML pages to be composed dynamically using a scripting engine that executes inside the IIS Web Server. Pages on the Web Server with `.asp` suffixes contain standard HTML tags interleaved with embedded scripts that are written in a simplified version of Visual Basic called VBScript. The reply page delivered to the Web Browser consists of the original HTML combined with the results of executing the scripts. For example, a very simple ASP page that returns the date and time on the Web Server, might take the following form:

```
<HTML>
<HEAD>
<TITLE>Remote Time</TITLE>
</HEAD>
<BODY>
You visited me on <% = Date %> at <% = Time %>
</BODY>
</HTML>
```

In this fragment, the VBScript interpreter replaces the text between the `<%` and `%>` delimiters by date and time strings, before returning the HTML page to the Web Brower.

Multi-line scripts can be formed by locating the `<%` and `%>` brackets on separate lines, or by using `<SCRIPT>` and `</SCRIPT>` HTML tags:

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">
...
</SCRIPT>
```

The attributes are necessary to identify VBScript as the scripting language, and to distinguish server-side scripts that run in the Web Server, from client-side scripts that run in the Web Browser. The `<%` and `%>` provide a more concise way of specifying server scripts, and we use them throughout the examples in this section.

In addition to the ability to generate HTML dynamically, server-side scripts can connect to any ActiveX Automation Server that is registered on the web server. In particular, an ASP page can connect to the PCRE Automation Server with a script such as:

```
<%
  Dim app
  Set app = Server.CreateObject("MAS.CRApplication.1")
  ...
  Set app = Nothing
%>
```

This fragment loads the PCRE Automation Server and stores a reference to the Application Object in the variable `app`. This gives us the ability to use the methods and properties to customize and produce reports. For example, the script

```
<%
  On Error Resume Next
  Dim app
  Set app = Server.CreateObject("MAS.CRApplication.1")
  app.ConnectToReportServer "flea.acme.com"
  Set report = app.OpenReport("C:\\SampleReports\\Box.rpt")
  report.PrintOut
  app.DisconnectFromReportServer
  Set app = Nothing
%>
```

shows how we can use the PCRE Automation Server to connect to a Report Server and generate a report. Notice how the script makes an independent connection to the Report Server with the call to `ConnectReportServer`. In a busy environment, this allows you to offload report generation to a dedicated machine, and preserve the performance of your Web Server.

In general, ASP-based report generation will use a configuration similar to that shown in Figure 5.6 on page 164. The sequence of events is as follows:

1. The client navigates to an ASP page with a URL which is typically something like

   `http://www.acme.com/SampleReports/sample.asp`

2. The IIS web server loads the `sample.asp` page and begins to compose the reply. During processing it encounters the `<%` and `%>` brackets and loads the VBScript interpreter to process the script.

3. As the script executes, it loads the PCRE Automation Server and then connects to the Report Server. The Report Server host can be specified in an HTML form sent with the document request in Step 1.

4. Customization calls to the Automation Server's methods are transmitted to the Report Server. If the report uses a database, the Report Engine Server will make an independent connection to the Database Server.

5. If the report has been generated in HTML or PDF and the Report Server does not run on the same machine as the Web Server, then the report should be retrieved to the Web Server.

6. Once the report has been generated, the script is disconnected from the Report Server and a reply is delivered to the browser. Typically, it contains a message confirming that the report was generated, together with a possible hyperlink to the report document.

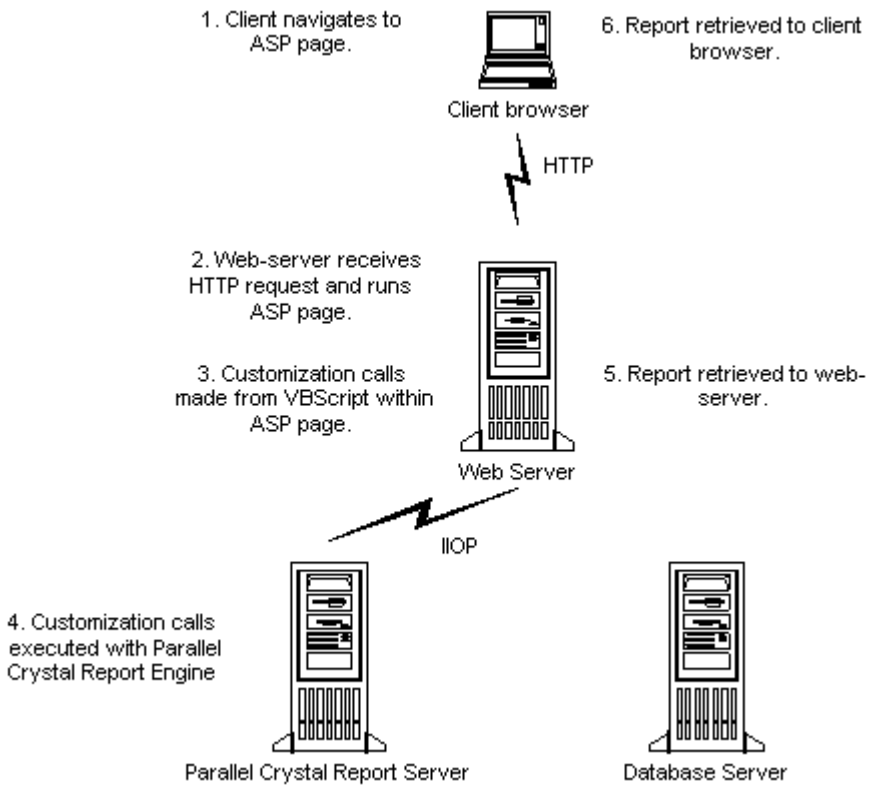

1. Client navigates to ASP page.
6. Report retrieved to client browser.
Client browser
HTTP
2. Web-server receives HTTP request and runs ASP page.
3. Customization calls made from VBScript within ASP page.
5. Report retrieved to web-server.
Web Server
IIOP
4. Customization calls executed with Parallel Crystal Report Engine
Parallel Crystal Report Server
Database Server

Figure 5.6

It is quite possible to have the IIS Web Server, the Report Server, and the Database Server all running on the same machine. The advantage of using Parallel Crystal is that it allows you to choose a suitable configuration, rather than imposing one on you.

In the next section, we'll describe some of the preparatory work you have to do to create an ASP project on the Web Server. Then we'll look at a simple script that extracts information from a form and uses it to generate a report in HTML. We cannot describe all the features of ASP or the VBScript language in this manual. We recommend you read a good introductory text such as the book by Francis et al[1], and then make the occasional excursion to the Platform SDK documentation in the MSDN.

[1] Brian Francis and others: Professional Active Server Pages 2.0, Wrox Press, 1998. ISBN 1-861001-26-6.

ASP Preliminaries

An ASP project is located within a sub-directory of your Web Server's root directory. If you have Microsoft IIS 4.0 installed, then the root will typically be located at `C:\InetPub\wwwroot`. An ASP project is usually organized within a sub-directory of the root such as `C:\InetPub\wwwroot\ASPDemos`. You should place your ASP pages in files with `.asp` suffixes in this directory, together with a single file called `globals.asa`. For simple projects, the `globals.asa` file should just contain the following empty procedures:

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">

Sub Session_OnStart
End Sub

Sub Session_OnEnd
End Sub

Sub Application_OnStart
End Sub

Sub Application_OnEnd
End Sub

</SCRIPT>
```

When the Web Server loads a page from an ASP project for the first time, it creates an Application Context to represent the project application, and a Session Context to represent the client. Thereafter, each new client that joins the Application gets its own Session Context. Application and Session Contexts maintain global and local state information that can be accessed by all the pages in a project. The information in an Application Context is accessible to all the clients, whereas the information in the Session Context is only accessible to the client that owns the Session. A Session Context is maintained until it is explicitly destroyed by the client, or until it times out. The Application Context is maintained until the Web Server is terminated.

When the ASP files have been created, we recommend that you use the Microsoft Management Console to isolate the execution of the embedded VBScript from the Web Server. You do this as follows:

1. From the Program Manager select Windows NT Server 4.0 Option Pack, and then select Microsoft Internet Information Server|Internet Service Manager.

2. Use the left hand scope pane to navigate to the directory containing your ASP files. If these are located in the directory `ASPdemo` in the Web Server's root, then this name will be visible in the scope pane.

3. Right click on the directory name and select the Properties field from the popup menu. The Properties Dialog shown in Figure 5.7 on page 166 appears.

4. Select the check box labeled "Run in separate memory space (isolated process)".

5. Press OK to implement the selection.

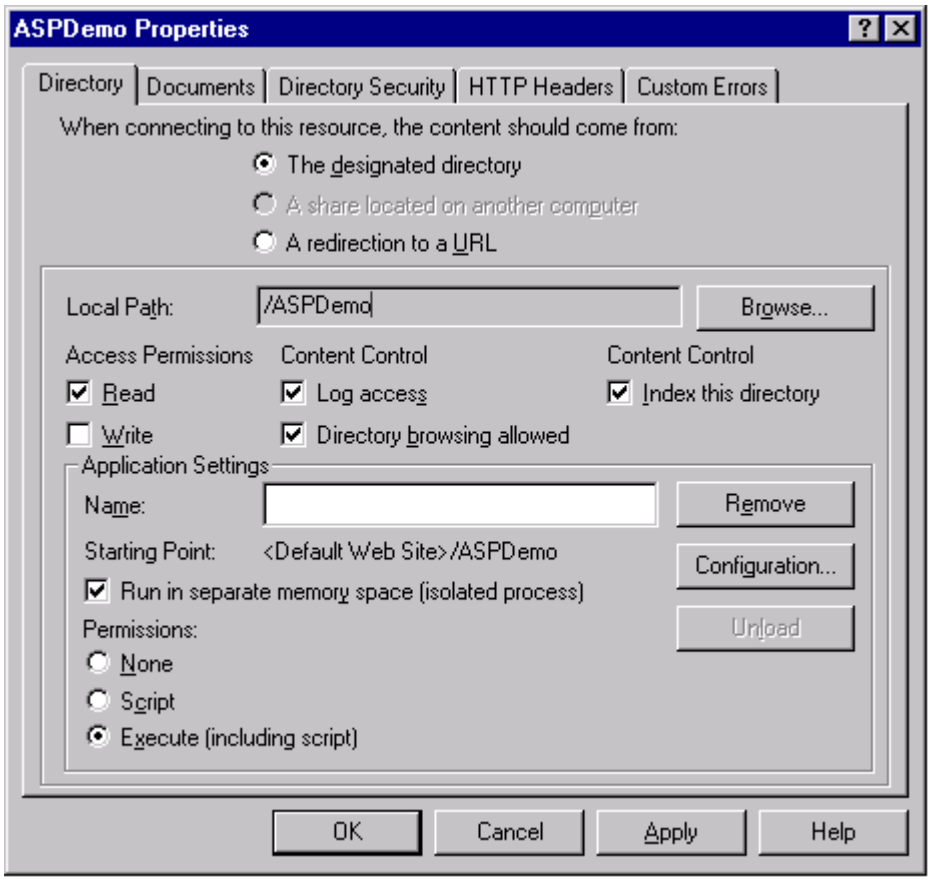


Figure 5.7

The effect of this selection is to run your scripts in a separate "Proxy server" process, rather than as part of the Web Server's process. If you are unfortunate enough to generate a fatal error during development, the Proxy Server may be terminated, but the Web Server will remain unharmed. This precaution is well worth taking!

Generating HTML Reports with ASP

In this section, we describe a simple ASP project that produces a report in HTML. The project comprises the following files located in the directory `ASPDemo` in the web server root directory:

```
InetPub\wwwroot\
                ASPDemo\
                        globals.asa
                        SimpleDemo.asp
                        SimpleAPICalls.asp
```

The `SimpleDemo.asp` file contains an HTML form that is used to identify the Report Server host, the name of the report file, and the name of the input file. When the browser is directed to this file with the URL

`http://www.acme.com/ASPDemo/SimpleDemo.asp`

the browser displays the page shown in Figure 5.8.

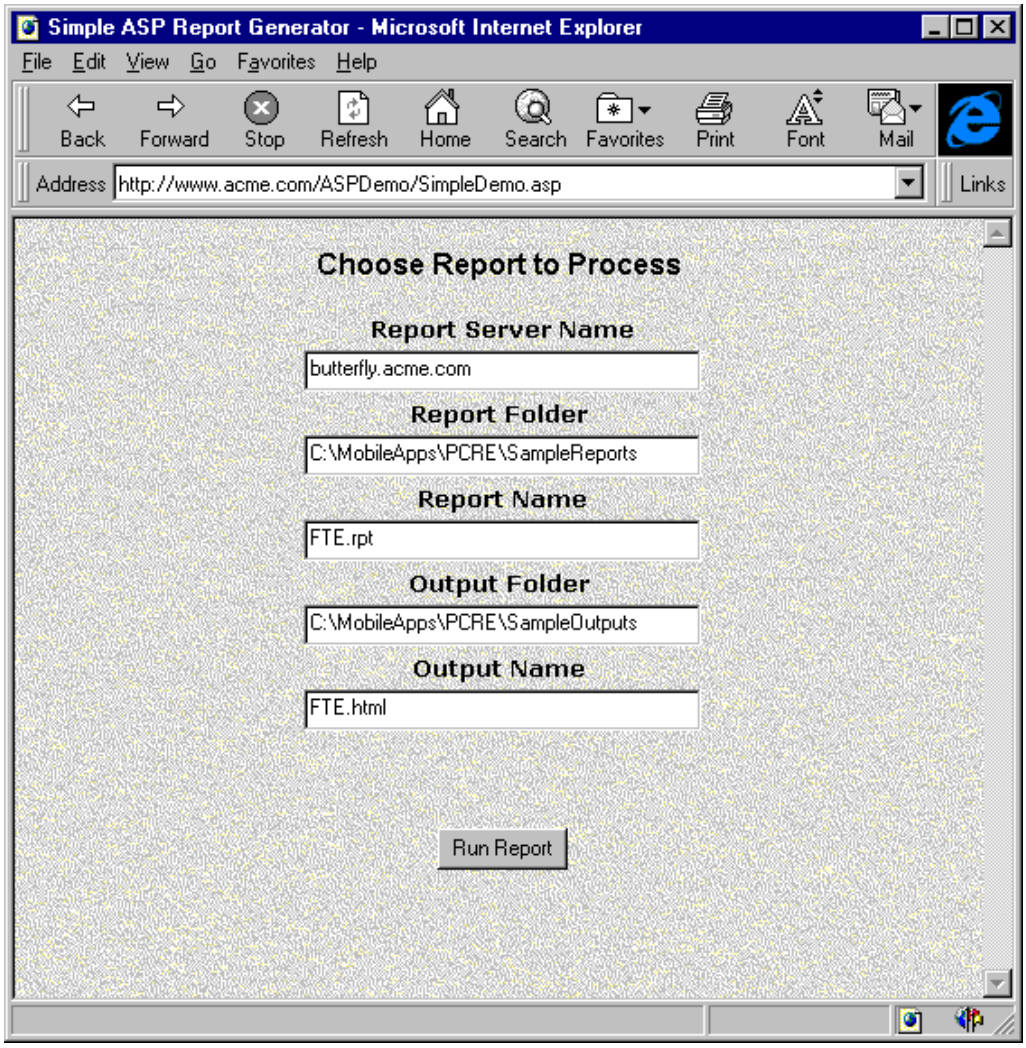


Figure 5.8

The contents of `SimpleDemo.asp` are shown below, with minor details of font and layout omitted:

```
<%@ LANGUAGE=VBScript %>
<HTML>
<HEAD>
<TITLE>Simple ASP Report Generator</TITLE>
</HEAD>
<BODY BACKGROUND="democolor.gif">
<CENTER>
<H2>Choose Report to Process</H2>
<FORM METHOD=post ACTION="SimpleAPICalls.asp">
<TABLE>
   <TR><TD><b>Report Server name</b></TD></TR>
   <TR><TD><INPUT TYPE=text NAME=ServerName SIZE=40
            VALUE="butterfly.acme.com"></TD>
   </TR>
   <TR><TD><b>Report Folder </b></TD></TR>
   <TR><TD><INPUT TYPE=text NAME=ReportFolder SIZE=40
            VALUE="C:\MobileApps\PCRE\SampleReports"></TD>
   </TR>
   <TR><TD><b>Report Name</b></TD></TR>
   <TR><TD><INPUT TYPE=text NAME=ReportName SIZE=40
            VALUE="FTE.rpt"></TD>
   </TR>
   <TR><TD><b>Output Folder </b></TD></TR>
   <TR><TD><INPUT TYPE=text NAME=OutputFolder SIZE=40
            VALUE="C:\MobileApps\PCRE\SampleOutputs"></TD>
   </TR>
   <TR><TD><b>Output Name </b></TD></TR>
   <TR><TD><INPUT TYPE=text NAME=OutputName SIZE=40
            VALUE="FTE.html"></font></TD>
   </TR>
</TABLE>
<TABLE>
    <TR><TD><INPUT TYPE=submit VALUE="Run Report"></TD></TR>
</TABLE>
</FORM>
</CENTER>
</BODY>
</HTML>
```

When the "Run Report" button is pressed, the browser posts the `<FORM>` to the Web Server. The Web Server analyzes the HTTP Post Request and passes it to the `SimpleAPICalls.asp page`. When the page is loaded, the VBScript interpreter recovers the entries in the form, runs the report, and returns the results page shown in Figure 5.9 on page 169.

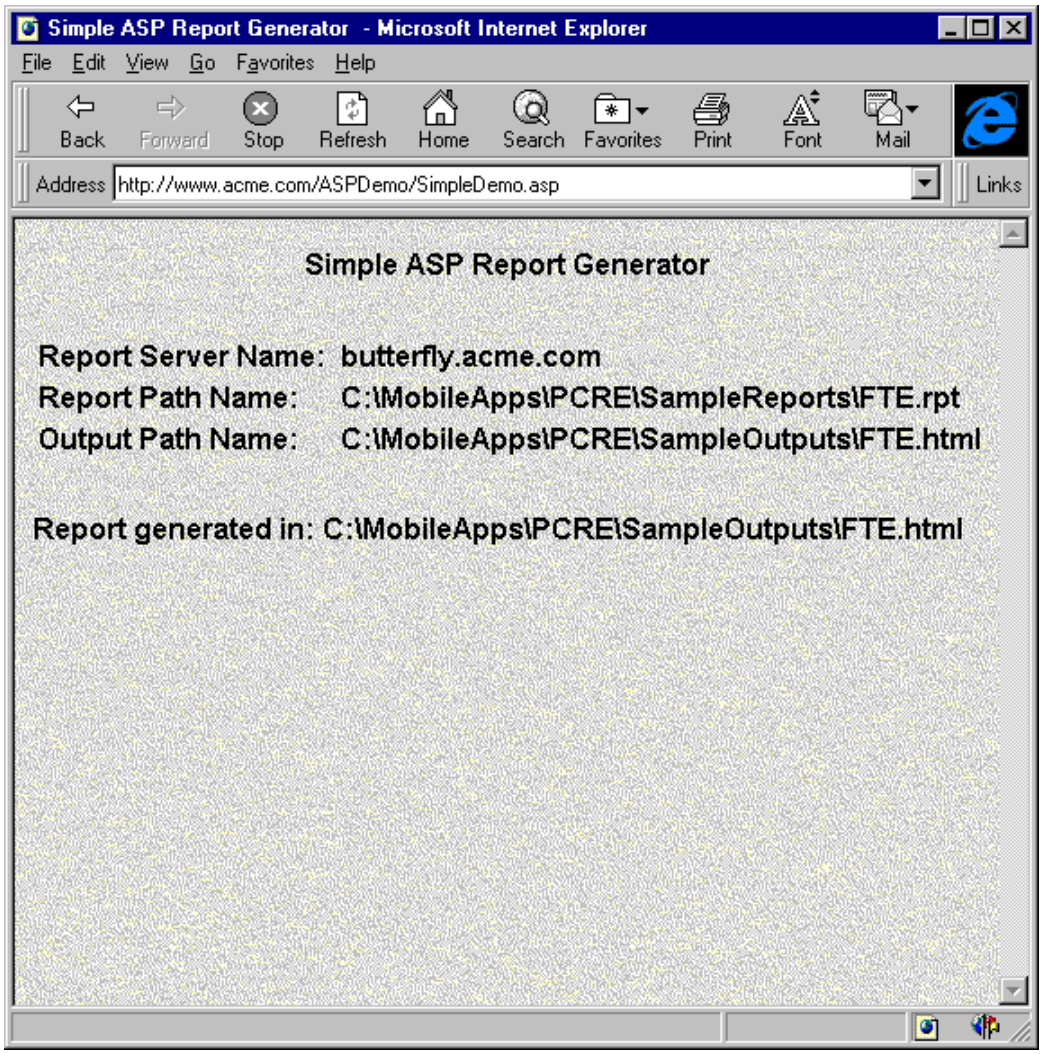

Simple ASP Report Generator - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites Print Font Mail
Address http://www.acme.com/ASPDemo/SimpleDemo.asp
Links
Simple ASP Report Generator
Report Server Name: butterfly.acme.com
Report Path Name: C:\MobileApps\PCRE\SampleReports\FTE.rpt
Output Path Name: C:\MobileApps\PCRE\SampleOutputs\FTE.html
Report generated in: C:\MobileApps\PCRE\SampleOutputs\FTE.html

Figure 5.9

The contents of the `SimpleAPICalls.asp` page are shown below. Note that the Visual Basic quote comments are included for the commentary that follows. The HTML `<-- ... -->` comments are too bulky to accommodate on a line.

The error handling for the code is for ASP 2.0. You can use the `Error.Ojbect` in ASP 3.0 for better error handling.

The response that is returned to the browser is derived from the static HTML sequences enclosed between conventional tags, and the dynamic sequences that are generated by the embedded VBScript. The lines in the page that are particularly important are highlighted in bold.

```
<%@ LANGUAGE=VBScript %>                                    ' 1.
<HTML>
<HEAD>
<TITLE> Simple ASP Report Generator </TITLE>
</HEAD>
<BODY>
<CENTER>                                                    ' 2.
<b>Simple ASP Report Generator</b>
</CENTER>
<br><br>

<%                                                          ' 3.
   Sub ReportError( msg )                                   ' 4.
      Response.write("SimpleAPICalls.asp: " & msg)
      bOK = False : Err.Clear                               ' 5.
   End Sub
%>
<%
   bOK = True : Err.Clear                                   ' 6.

   ReportServer = Request.Form("ServerName")                ' 7.
   ReportFolder = Request.Form("ReportFolder")
   ReportName = Request.Form("ReportName")
   ReportPath = ReportFolder + "\" + ReportName
   OutputFolder = Request.Form("OutputFolder")
   OutputName = Request.Form("OutputName")
   OutputPath = OutputFolder + "\" + OutputName
%>

<TABLE>                                                     ' 8.
   <TR><TD><b>Report Server name: </b></TD>
       <TD><b><%=ReportServer%></b></TD>
   </TR>
   <TR><TD><b>Report Path Name: </b></TD>
       <TD><b><%=ReportPath%></b></TD>
   </TR>
   <TR><TD><b>Output Path Name: </b></TD>
       <TD><b><%=OutputPath%></b></TD>
   </TR>
</TABLE>

<%                                                          ' 9.
   On Error Resume Next                                     ' 10.
   Dim PCREApp
   Set PCREApp = Server.CreateObject("MAS.CRApplication.1") ' 11.
   If Not IsObject(PCREApp) Then                            ' 12.
      ReportError "Report Server not available"
   End If
```

Example 5.4 (cont. on next page)

```
   If bOK Then                                                ' 13.
      PCREApp.ConnectToReportServer(ReportServer)
      If Err.Number <> 0 Then
         ReportError "Cannot connect to Report Server on: " & _
                     ReportServer
      End If
   End If


   If bOK Then                                                ' 14.
      Dim Report
      Set Report = PCREApp.OpenReport(ReportPath)
      If Not IsObject(Report) Then
         ReportError "Cannot open report file: " & ReportPath
      End If
   End If

   If bOK Then                                                ' 15.
      Report.ExportToHTML OutputPath
      If Err.Number <> 0 Then
         ReportError "Cannot export report to : " & OutputPath
      End If
   End If

   If bOK Then                                                ' 16.
      Response.write("<br><br><b>Report generated in: " & _
                     OutputPath & "</b>")
   End If

   If IsObject(PCREApp) Then                                  ' 17.
      PCREApp.DisconnectFromReportServer
   End If
   Set PCREApp = Nothing                                      ' 18.
%>
</BODY>
</HTML>
```

Example 5.5

Each numbered paragraph below corresponds to the same numbered code statement

1. This line appears at the start of most ASP pages. It identifies VBScript as the server-side scripting language for the whole page.

2. The text between the `<CENTER>` and `</CENTER>` tags forms the title for the response.

3. Lines of VBScript are distinguished from HTML by being enclosed in `<%` and `%>` brackets.

4. The procedure `ReportError` is responsible for composing an error message and embedding it in the response. This is done by the call to the `write` method of the built-in object `Response`. Each call to write appends a line to the response returned to the browser.

5. This line sets a Boolean variable `bOK` to `False` and clears the built-in error indicator called `Err`. `bOK` is used to control the flow of execution through the report generation part of the script.

6. This line implicitly declares the variable called `bOK` and clears the current error indicator. All variables in VBScript have the Visual Basic type `Variant`, so you need not use explicit `Dim` statements.

7. This sequence of statements shows how VBScript uses the built-in object `Request` to retrieve the strings entered into the HTML form. Each entry is recovered using an equivalently named property of the `Request` object. The entries are treated as `Variants` containing strings, and therefore we can combine them using the Visual Basic string concatenation operator &.

8. This sequence of static HTML formats a table containing the Report Server host name, the path name of the report file, and the path name of the HTML report to be generated.

9. The script between `<%` and `%>` brackets generates the report using the highlighted methods of the PCRE Automation Server.

10. This line is the most important on the whole page. When you make calls to the PCRE Automation Server, the VBScript interpreter handles the low-level COM interface internally. However, if a method returns a failure result code, the interpreter stops and delivers the code back to the client in the form of a hexadecimal string. The `On Error Resume Next` statement allows us to override the default error handler and provide our own error message. However, it also means that after every method call, the result code must be tested using the default `Err` object. We'll consider various options for error handlers in the next section.

11. This statement connects the script as a client of the PCRE Automation Server. `CreateObject` is a method of the built-in `Server` object that loads the Server DLL and stores a reference to the Application Object in the variable `PCREApp`. The string argument to `CreateObject` is the Automation Server's `PROGID`. It corresponds to an entry in the System Registry that should be established at installation time. We'll return to this topic in Chapter 7. Notice that the `bOK` flag is used to control execution of the remainder of the script. If this variable becomes `False`, then control falls through to Step 17.

12. If the call to `CreateObject` fails, then the result will not be a valid reference. The `IsObject` function provides a way to test for this possibility. If the test fails, the error message is inserted into the response by the call to `ReportError`.

13. Once a connection to the Automation Server is established, the script calls the `ConnectToReportServer` method, passing the host name as an argument. It is vital to check the result, since the name supplied might not be valid, or the Report Server might not be running on the nominated host.

14. If the connection to the Report Server is established, the `OpenReport` method is used to return an instance of the Report Object for the report specified by the `ReportPath` argument. Once more, it is necessary to use `IsObject` to test the result, since the method will fail if the file specified by the argument does not exist.

15. The `ExportToHTML` method of the Report Object generates the report and "exports" it to the file specified by the `OutputPath` argument. Note that the HTML file is generated on the Report Server, and not on the Web Server where the script is running. We'll discuss various options for retrieving the file below.

16. If the call to `ExportToHTML` executes successfully, a final confirmation message is written to the response page.

17. It is extremely important to disconnect from the Report Server before the script terminates. However, we test the `PCREApp` reference for validity first of all. Note that inspecting `bOK` is not good enough because we could have a situation in which `PCREApp` was valid but `bOK` was `False` because a subsequent call to the Report Server failed.

18. The call to set `PCREApp` to `Nothing` releases resources in the PCRE Automation Server.

In the following sections we'll discuss various issues relating to report retrieval and error handling, and show how the script can be trivially changed to generate reports in other formats.

Retrieving Reports with ASP

If the Report Server and the Web Server run on different machines, then you need to retrieve the generated report back to a suitable location the Web Server before you can view it in your browser. If you return this location as a hyperlink in the response, then you can navigate directly to the report.

The modifications that are required to the example's ASP pages are as follows:

1. In the page `SimpleDemo.asp`, change the form so that the Output Folder entry box holds the name of a sub-directory called `SampleOutputs` rather than the full path name `C:\MobileApps\pcre\SampleOutputs`. When the report HTML file is retrieved, it will be placed in the `SampleOutputs` sub-directory of the Web Server's root directory.

To get the report's HTML file back to the Web Server, we use the `ReportRetrieval` method in the Automation Server. To use this method, the following changes to the `SimpleAPICalls.asp` page are required:

2. At Step 7 in the example insert the line

```
WebRoot = "C:\InetPub\wwwroot"
```

   If your web server has a different root directory, then insert the appropriate value.

3. Insert the following new code between the end of Step 14 and the start of Step 15:

```
If bOK Then
   PCREApp.RetrieveMode = True
   If Err.Number <> 0 Then
      ReportError "RetrieveMode: cannot enable"
   End If
End If
```

   This code sets the `RetrieveMode` property of the Application Object to `True` which enables automatic retrieval of the report once it is generated. However, when you enable report retrieval, you must specify the location of the file on the Web Server rather than on the Report Server. Therefore, we must change the call to `ExportToHTML` to

```
Report.ExportToHTML WebRoot & "\" & OutputPath
```

   If `WebRoot` is set as shown, `OutputFolder` is `"SampleOutputs"` and `OutputName` is `"FTE.rpt"`, then the HTML report will be retrieved to the file

```
C:\InetPub\wwwroot\SampleOutputs\FTE.html
```

4. Rather than return this path name in the response, we can return a hyperlink so that the user can navigate directly to the `FTE.html` file. We can do this with the following modification to Step 16:

```
If bOK Then
  Anchor = "<A HREF=" & Chr(34) & "/" & OutputPath & Chr(34) & ">"
  Response.write("<br><br><b>Report generated in: " & _
                 Anchor & OutputName & "</A></b>")
End If
```

   This generates the following HTML and writes it to the response sent to the user:

```
<br><br><b><A HREF="/SampleOutputs/FTE.html">FTE.html</A></b>
```

   When the user clicks on the hyperlink, the `FTE.html` file will be displayed in the browser.

5. If you want the report to open automatically, without an intermediate page, you can do it by using various page redirection techniques, like using `Response.Redirect` in ASP 2.0 or `Server.Transfer` in ASP 3.0.

   If you are using ASP 2.0 and you do not want to use `Response.Redirect`, then you can use a javascript client side script to open this new page.

The most important thing to remember when using the Report Retrieval feature is that the path name for the generated report must be specified on the Web Server, and not on the Report Server. If you want the file to be directly available for browsing, then you must retrieve it to a file store location within the Web Server's root directory.

Generating PDF and PostScript Reports with ASP

The `ICRReport` interface provides several methods that make it easy for you to generate reports in other formats. In each case, the call to `ExportToHTML` in Step 15 is replaced with one that generates the report in the appropriate format. In particular:

- To generate a report in PDF and save the results, replace the call to `ExportToHTML` with the following call to `OutputToPDF`:

  ```
  report.OutputToPDF OutputPath
  ```

  When the method returns, the PDF report will be stored in the file specified by the `OutputPath`. You should use a `.pdf` subscript to identify PDF files.

- To generate a report and send the results to a printer, replace the call to `ExportToHTML` with the following call to `PrintOut`:

  ```
  report.PrintOut
  ```

  When the method returns, the report has been generated but printing will not necessarily be complete.

- To generate a report in PostScript and save it in a file, replace the call to `ExportToHTML` with the following call to `OutputToPS`:

  ```
  report.OutputToPS OutputPath
  ```

  The report will be generated in the file identified by the `OutputPath`. Note that you cannot depend upon the file being completely generated when you return from the method call. This is because the file must be generated by the Windows Printer Spooler that does not run synchronously with the Report Engine Server.

All these methods cause the print job to be started, so you must complete all necessary customization in previous method calls. You can also retrieve the files back to the Web Server by setting the `RetrieveMode` property to `TRUE` and setting `OutputPath` for the Web Server rather than the Report Server.

Handling Errors from ASP 2.0

When an error occurs in the Automation Server, it stores an error message internally and returns a generic COM error called `E_FAIL`. Unfortunately, when the VBScript interpreter detects the error, it writes the hexadecimal value of `E_FAIL` together with the COM description string

```
"Unspecified error"
```

to the response page, and terminates without disconnecting the client from the Report Server.

In order to handle errors satisfactorily in VBScript, you must adhere to the following guidelines:

1. Insert an `On Error Resume Next` statement at the start of the script. When an error occurs, the VBScript interpreter will return control to the statement following the call to the Automation Server method or property.

2. After every method or property call that can return a COM error, check the `Number` property of the built-in `Err` object. If the value is non-zero, then an error has occurred.

3. Call the `LastErrorString` property of the `ICRApplication` or `ICRReport` interface to return the Automation Server error message.

4. Always ensure that if you have called `ConnectToReportServer`, you make a corresponding call to `DisconnectFromReportServer`.

5. Use `IsObject` to check object references returned by the Automation Server are valid.

For example, here is a slightly modified fragment from the example that uses the `LastErrorString` property to access the error message:

```
If bOK Then
   Dim Report
   Set Report = PCREApp.OpenReport(ReportPath)
   If Not IsObject(Report) Then
      ReportError "OpenReport: " & PCREApp.LastErrorString
   End If
End If

If bOK Then
   Report.ExportToHTML(OutputPath)
   If Err.Number <> 0 Then
      ReportError "ExportToHTML: " & Report.LastErrorString
   End If
End If
```

If the error does not occur in an Automation Server method or property, then you can call the `Err.Description` property that will return a generic COM error message. For example:

```
Dim PCREApp
Set PCREApp = Server.CreateObject("MAS.CRApplication.1")
If Not IsObject(PCREApp) Then
   ReportError "CreateObject: ", Err.Description
End If
```

Sometimes, it may be useful to report particularly serious errors with a message box. Clearly, if the message is displayed on the Web Server, the client can neither view nor dismiss it. However, you can produce a message box in the client's browser by modifying the `ReportError` procedure in the example as follows :

```
Sub ReportError( msg )
   Response.write("<SCRIPT LANGUAGE=" & Chr(34) & "VBScript" & _
               Chr(34) & ">")
   Response.write("Alert " & Chr(34) & msg & Chr(34))
   Response.write("</SCRIPT>")
   bOK = False : Err.Clear
End Sub
```

This piece of trickery generates the following script that is executed in the browser when it receives the response:

```
<SCRIPT LANGUAGE="VBScript">
Alert "<contents of msg>"
</SCRIPT>
```

The actual argument to `Alert` is the `msg` string enclosed within double quotes. Notice the use of the _ continuation marker to allow a VBScript statement to be continued on the next line, and the : separator that allows multiple statements to occupy the same line.

Note: The Error.Object in ASP 3.0 may be used for better error handling.

Trouble Shooting Simple ASP Clients

In this section we'll take a look at the most common errors that occur when programming simple clients using ASP. We'll assume that errors are reported by accessing the appropriate `GetLastErrorString` property in the Automation Server.

- The call to the Application Object method `ConnectToReportServer` may fail and report the error:

  ```
  "No connection to a Server Report Engine has been made"
  ```

  There are two possible reasons for this error: either the argument string does not specify a valid host name, or the Parallel Crystal Report Server was not running on the specified host. Check that the string you supplied is either a valid host domain name or IP address, and then check that Parallel Crystal is installed and running on that host. If the problem persists, check Chapter 7, Trouble Shooting ActiveX Clients.

- The call to the Application Object method `OpenReport` may fail and report the error:

  ```
  "Invalid file name"
  ```

  This error occurs when the report file specified by the argument string does not exist on the Report Server. Check the location of the file and then supply its full path name to `OpenReport`.

- The call to the Report Object method `ExportToHTML` may fail and report the error:

  ```
  "Disk full"
  ```

  This highly misleading message is actually returned by the Report Engine and means that it was unable to create the HTML file to hold the generated report. It does *not* mean your disk is full! Check the argument string for `ExportToHTML` specifies a valid drive and path name, and that access permissions on the Report Server file store allow the file to be created.

- The call to the Report Object method `PrintOut` may fail and report the error:

  ```
  "Cancelled by user"
  ```

  This message is returned by the Report Engine when it is unable to locate a suitable printer. This is either because the printer has become inaccessible because of network problems, or because the printer settings inserted into your report at design time, are no longer valid. You may need assistance from either your System Administrator, or Dynalivery Technical Support Staff to fix this problem.

Chapter 6
Using the PCRE Automation Server

This chapter provides detailed information on how Automation Server clients access and use the Parallel Crystal Report Server.

The chapter is divided into the following sections

- The section entitled Automation Server Clients describes how clients connect and disconnect to and from the Report Server via the Automation Server. The relevant methods of the `ICRApplication` interface are described in detail for clients using Visual C++, Visual Basic, VBScript and Visual J++, and for local and remote connections.

- The section entitled Report Generation describes hierarchy of Automation Objects in the PCRE Object Model. Each object is described by listing the methods and properties of its interface and giving simple examples showing how objects are created and methods and properties are invoked. For brevity, these examples are given in Visual Basic, and you are referred to the Automation Server's online documentation for the syntax for other languages.

- The section entitled Parallel Crystal Configuration Server describes how clients can retrieve configuration information such as the installation path names of various directories. The information currently returned is primarily useful for sampling the tutorial examples provided with Parallel Crystal.

- The section entitled Parallel Crystal Report Retrieval describes how reports generated on a remote Report Server can be retrieved to the client's machine. Web-based clients such as ASP pages can use this service to pull a report back to the Web Server, before returning a suitable hyperlink in the response page.

- The section entitled Parallel Crystal Load Balancer describes the load balancer service that allows client connections to be distributed across a collection of Report Servers. When clients connect in "automatic mode" the Load Balancer uses dynamic load monitoring to select an appropriate Report Server. However, clients can also request load monitoring information to be returned by the Load Balancer, enabling the final section of the Report Server to be made "manually".

Automation Server Clients

In this section, we'll describe how ActiveX clients use the Automation Server to connect and disconnect to and from the Parallel Crystal Report Server. We'll use the term "automation client" to refer to any client written in Visual Basic, VBScript, Visual J++, or Visual C++.

Automation client connection and disconnection is essentially a four-step process:

1. The client "connects" to the Automation Server. In practice, "connects" means that the Automation Server DLL is loaded into the client's process; an instance of the Application Object is created; and a reference to the object's `ICRApplication` interface is returned to the client. No network activity is involved at this stage.

2. The client calls the `ConnectToReportServer` method of the `ICRApplication` interface to connect to the Parallel Crystal Report Server. If the Report Server runs on the same machine as the client, the connection will cross a process boundary. However, if the Report Server runs on a different machine, the connection will cross whatever networks are involved. When the connection is established, the client is connected to a Report Engine Server that does report customization and generation.

3. The client calls the `DisconnectFromReportServer` method to disconnect from the Report Server. When the connection is severed, the client's Report Engine Server is terminated and memory resources in the client are freed.

4. The client "disconnects" from the Automation Server by calling the `Release` method of the `ICRApplication` interface. This causes residual memory to be recovered by the Automation Server, and may cause the DLL to be unloaded from the client process. In practice, only C++ clients using the "bare" COM API need to make an explicit call to `Release`. The COM interfaces provided in Visual Basic, VBScript and Visual J++ release the interface internally.

It is very important for clients to call `DisconnectFromReportServer` to terminate their Report Engine Servers, and to release client-side resources.

Connecting to the Automation Server

The manner in which Automation Server connections are made depends upon the choice of client language. In particular:

- If you use C++ with the "bare" COM API, the Application Object reference is returned by calling `CoCreateInstance`:

```
HRESULT hr;
ICRApplication *pApp;
hr = CoCreateInstance(CLSID_CRApplication, NULL,
                      CLSCTX_INPROC_SERVER, IID_ICRApplication,
                      reinterpret_cast<void **>(&pApp));
```

- If you use Visual C++ COM support classes, the Application Object reference is returned by calling the `ICRApplicationPtr` constructor:

```
ICRApplicationPtr pApp;
pApp(__uuidof(CRApplication));
```

- If you use Visual J++ and have imported the Automation Server Type Library with the Java Type Library wizard, the Application Object reference is returned by creating an instance of the Java `CRApplication` wrapper class and casting the result to the `ICRApplication` interface:

```
ICRApplication app = (ICRApplication)new CRApplication();
```

- If you use Visual Basic and have imported the Automation Server Type Library with the Project References dialog, the Application Object reference is returned by creating an instance of the `CRApplication` co-class and storing the result in an object reference variable declared with type `ICRApplication`:

```
Dim app As ICRApplication
Set app = New CRApplication
```

- If you use VBScript with ASP, the Application Object reference is returned by calling CreateObject and specifying the Automation Server's PROGID:

```
Dim app
Set app = Server.CreateObject("MAS.CRApplication.1")
```

Each of these calls relies on the Automation Server DLL being installed on the client machine and appropriate information set up in the registry as follows.

The Automation Server DLL is called `cpemas1.dll` and is normally located in the installation directory `C:\MobileApps\pcre\bin`. The full path name of the DLL is entered into your client machine registry at installation time by the command

```
regsvr32 cpemas1.dll
```

When the command is executed it creates the following entries in the registry hive `HKEY_CLASSES_ROOT`:

```
MAS.CRApplication\
    CurVer\                "MAS.CRApplication.1"
    CLSID\                 {81B34614-B2B4-11D1-8C38-000000000000}
MAS.CRApplication.1\
    CLSID\                 {81B34614-B2B4-11D1-8C38-000000000000}

{81B34614-B2B4-11D1-8C38-000000000000}\
    InprocServer32\           C:\MobileApps\pcre\bin\cpemas1.dll
       ThreadingModel         "Apartment"
    ProgID\                   "MAS.CRApplication.1"
    VersionIndependentProgID\ "MAS.CRApplication"
```

The external environments that run your automation controller use these entries in the following ways:

- `MAS.Application` and `MAS.Application.1` are called PROGIDs, and allow VBScript and JScript controllers to locate the automation server DLL with the `CreateObject`.

- `{81B34614-B2B4-11D1-8C38-000000000000}` is the CLSID that uniquely identifies the application object's co-class. The co-class implements the interface `ICRApplication` that forms the root of the object hierarchy. Collectively, the registry entries provide a way to map from PROGIDs to CLSIDs and back again. Visual C++ automation controllers can use the CLSID directly, but Visual Basic controllers must locate the PROGID first, and then use that to locate the CLSID.

- `InprocServer32` identifies the full path of the automation server DLL and indicates it is loaded into the same process address space as your client. This means that calls on the automation objects' methods and properties are efficient since they do not cross thread, process, or machine boundaries.

- `ThreadingModel` identifies the threading characteristics of the automation server. The value `Apartment` allows a single process to run multiple copies of the automation server concurrently, provided each executes on a separate thread. In environments such as a Web Server that maintain thread-pools, the value `Apartment` additionally ensures that once a thread is assigned to run the Automation Server, it is only ever run on that thread.

If the Automation Server DLL has not been installed on the client machine, or the registry entries have been corrupted or accidentally destroyed, or you have changed the file store location of the DLL, then each of the calls to create the Application Object will fail with a COM error whose `HRESULT` code is `0x80040154`. If you use one of the COM error reporting mechanisms to print the error description, you will get the message

```
"Class not registered"
```

which indicates that the COM runtime layer was unable to locate the `CRApplication`'s in-process server DLL. The Visual J++ runtime will also provide its own error message:

```
"java.lang.UnsatisfiedLinkError: Failed creation of
 cpemas1/CRApplication because CLSID { ... } is not
 properly registered"
```

To rectify this error, you should first unregister the Automation Server DLL with the command

```
regsvr32 /u cpemas1.dll
```

and then re-register it with the command

```
regsvr32 cpemas1.dll
```

Note that the DLL must also be on your PATH for these commands to work.

Connecting to the Report Server

Once you have a reference to the Application Object's `ICRApplication` interface, you can connect to a Report Server by calling the method `ConnectToReportServer`. This method is declared as follows:

```
// Visual C++
HRESULT ConnectToReportServer( BSTR host )

' Visual Basic
Sub ConnectToReportServer( ByVal host As String )

// Visual J++
void ConnectToReportServer( java.lang.String host )
```

The argument string host specifies the Report Server to which your client will connect. It may take the form of either a symbolic host computer name, or a dotted computer IP address. For example

```
// Visual C++
app->ConnectToReportServer("mantis");

' Visual Basic
app.ConnectToReportServer("example.host.com")

// Visual J++
app.ConnectToReportServer("123.456.654.321")
```

The method will return a COM error if `host` does not specify a valid computer name, or if a Parallel Crystal Report Server is not running on the nominated machine.

If you supply an empty string for the host name, then your client will be connected to a Report Server "somewhere" on your network. This is okay if you know that there is only one Report Server and you don't necessarily care about its location. However, if you have multiple Report Servers and you leave the host unspecified, you cannot predict which server the `ConnectToReportServer` method will select. Therefore, you must take steps to ensure that your report files are accessible on all of the Report Servers to which you could possibly connect. This may not be easy, so we recommend that you specify a host whenever possible.

A single call to `ConnectToReportServer` is normally sufficient to establish a connection to a Report Server running on your local network. However, if the Report Server runs on a remote network you may need to set the `OSAgentAddress` property. The OSAgent is a Report Server component that services client connection requests. OSAgents are normally reached by UDP[1] broadcasts made from the client. However, when the Report Server is remote, the UDP mechanism does not work, and you must supply the IP address to enable the connection to be established.

[1] UDP is the Internet's User Datagram Protocol.

The `OSAgentAddress` property is declared as follows:

```
// Visual C++
HRESULT put_OSAgentAddress( BSTR ipAddr )

' Visual Basic
Property Let OSAgentAddress( ByVal ipAddr As String )

// Visual J++
void putOSAgentAddress( java.lang.String ipAddr )
```

You must set this property *before* calling `ConnectToReportServer`. For example:

```
// Visual C++
app->OSAgentAddress = "123.456.654.321";
app->ConnectToReportServer("example.host.com");

' Visual Basic
app.OSAgentAddress = "123.456.654.321"
app.ConnectToReportServer "example.host.com"

// Visual J++
app.putOSAgentAddress("123.456.654.321");
app.ConnectToReportServer("example.host.com");
```

Note that the arguments to `OSAgentAddress` and `ConnectToReportServer` need not specify the same host address. This is because multiple Report Servers can be serviced by a single OSAgent. You should check the required addresses with your System Administrator.

When the call to `ConnectToReportServer` returns successfully, your client is connected to a Report Engine Server running on the nominated host. The Report Engine is allocated exclusively to your client, and performs customization and report generation in response to calls on the methods and properties of the `ICRApplication` and `ICRReport` interfaces. The Report Engine Server always runs in a stand-alone server process.

Disconnecting from the Report Server

You must call the `ICRApplication` method `DisconnectFromReportServer` in order to terminate your Report Engine Server and release corresponding memory resources in the Automation Server. This method is declared as follows:

```
// Visual C++
HRESULT DisconnectFromReportServer( void )

' Visual Basic
Sub DisconnectFromReportServer()

// Visual J++
void DisconnectFromReportServer()
```

This method never returns a COM fail code. Typically, you call `DisconnectFromReportServer` as follows:

```
// Visual C++
app->DisconnectFromReportServer();

' Visual Basic
app.DisconnectFromReportServer

// Visual J++
app.DisconnectFromReportServer();
```

`DisconnectFromReportServer` also recovers memory resources in the Automation Server. However, to be effective Visual C++ clients must ensure that all interfaces to Automation Objects have been released manually or via smart pointers, the disconnect call is made. Visual Basic and Visual J++ clients don't need to worry about memory management because the underlying COM interface handles this automatically.

Disconnecting from the Automation Server

Your client is disconnected from the Automation Server when the `ICRApplication` interface is released. In practice, this is performed automatically by all clients except those using "bare" COM from C++. In that case, following the call to `DisconnectFromReportServer`, you should call `Release` as follows:

```
app->DisconnectFromReportServer();
app->Release();
```

This causes a final release of memory resources in the Automation Server.

Report Generation

The Parallel Crystal Automation Server provides a collection of Automation Objects whose interfaces may be used to customize and generate reports. Collectively, they are a subset of those provided in Seagate's Crystal Object Library since they omit functionality relating to report design. However, they include extra properties and methods for controlling aspects of the Parallel Crystal Report Server such as Client Connections, Report Retrieval, and Load Balancing. These topics are described in more detail later in this Chapter.

The interfaces in the Automation Server form a hierarchy called the PCRE Object Model which is shown in Figure 5.1 on page 125 in the previous chapter, and is reproduced here in a more compact form:

```
ICRApplication
   ICRGlobalOptions
   ICRReportServers
   ICRServer
   ICRReport
      ICRArea(s)
         ICRAreaOptions
         ICRGroupAreaOptions
         ICRSection(s)
            ICRSectionOptions
            ICRReportObject(s)
                ICRSubReportObject
                ICRGraphObject
                ICRTextObject
      ICRSection(s)
         ICRSectionOptions
         ICRReportObject(s)
            ICRSubReportObject
      ICRSortField(s)
      ICRDataBase
         ICRDataBaseTable(s)
            ICRDataBaseFieldDefinition(s)
         ICRDataBaseParameter(s)
      ICRPrintingStatus
      ICRPrinterInfo
      ICRReportSummaryInfo
      ICRExportOptions
      ICRReportOptions
      ICRFormulaFieldDefinition(s)
      ICRSummaryFieldDefinition(s)
      ICRParameterFieldDefinition(s)
      ICRPDFOutputOptions
         ICRPDFImage
   ICRReportSaver(s)
```

The nature of the hierarchy and its properties and methods conform to Microsoft's guidelines for automating document-oriented applications.

These guidelines apply to the PCRE Object Model

- `ICRApplication` is the interface on the root Application Object created when your client first connects to the Automation Server. The `OpenReport`, `ServerSettings` and `Options` methods/properties on this interface allow you to create Report, Server and Global Options Objects and to acquire their respective interfaces. `ICRReport`, `ICRServer`, and `ICRGlobalOptions`.

- Each object is a parent of the objects created by its methods or properties. For example, the Application Object is a parent of the Report, Server and Global Options Objects.

- Every interface has two properties `Parent` and `Application` that respectively return the interface of the parent and root Application objects. Given an interface such as `ICRDataBase`, you use its `Parent` property to locate the interface on its parent Report Object, and its `Application` property to locate the interface on the Application Object. You use the `Parameters` and `Tables` properties to create child DataBase Parameters and DataBaseTables Objects.

- The upward links in the hierarchy guarantee that a parent object can never be destroyed until all its children are destroyed. So the collection Automation Object instances is created from the top down, and dismantled from the bottom up.

- An automation client MUST release every interface it acquires. If you use Visual Basic, VBScript, JScript or Visual J++, interfaces are released automatically. If you use Visual C++ you must release the interfaces yourself.

- Interface properties and methods are specified with automation types `BSTR`, `VARIANT`, and `VARIANT_BOOL`.

- Every interface in the PCRE Object Model is a "dual" interface providing access to its methods and properties by both custom vtables and the dispatch mechanism.

- All methods and properties that create child objects initially return a reference to the `IDispatch` interface on the object. You can access the custom interface by calling `QueryInterface` with the required interface identifier. Conversions from dispatch to custom interfaces are made automatically by Visual Basic, by casts in Visual J++, and by smart pointers in Visual C++. VBScript and JScript clients are constrained by these languages to use the dispatch interface.

- Structurally, a report is modeled as a set of collections of various kinds. In the diagram above, each interface post-fixed with a (s) represents a collection. All collection interfaces provide a standard set of properties and methods that allow you to iterate through their members. Special consideration is given to Visual Basic that has a `For Each` statement that is explicitly designed to work with automation collections.

All automation clients acquire a reference to the root `ICRApplication` interface first of all, and then call `OpenReport` to create one or more Report Objects on named report files. Thereafter, the methods and properties of the `ICRReport` interface are used to customize and generate each of the reports. Roughly speaking, acquiring an `ICRApplication` reference corresponds to opening a Print Engine on the Report Server, and calling `OpenReport` corresponds to opening a print job.

In the remaining sections, we provide top-level descriptions of each Automation Object and its interface. The properties are divided into those that allow you to customize the report, and those that allow you to navigate through the object hierarchy. Each interface definition is accompanied by code samples that show how to acquire the interface, how to call its methods and properties, and how to iterate through collections. For brevity, these samples are given using Visual Basic syntax. They can be adapted to other Automation Client languages as follows:

- Visual Basic

```
Dim report As ICRReport
Set report = app.OpenReport "C:\SampleReports\Box.rpt"
```

- VBScript

```
Dim report
Set report = app.OpenReport "C:\SampleReports\Box.rpt"
```

  To convert to VBScript, remove the `As` clause from each `Dim` statement. Use the same syntax for calling methods and properties.

- Visual J++

```
ICRReport report;
report = (ICRReport)app.OpenReport("C:\SampleReports\Box.rpt");
```

  To convert to Visual J++, use the interface name to declare an interface reference variable. When calling a property that returns an interface, cast the property result using the interface name as shown above. Insert left and right parentheses for all method and property calls.

- Visual C++

```
ICRReport *report;
report = static_cast<ICRReport *>
            (app->OpenReport("C:\\SampleReport\\Box.rpt"));
```

  To convert to Visual C++, use the interface name to define an interface pointer variable. When calling a property that returns an interface, use a static_cast to cast the result to the interface pointer as shown. Insert left and right parentheses for all method and property calls.

The interface descriptions are only intended as summaries. You should consult the PCRE Automation Server online documentation for detailed descriptions of the parameter types, result codes and functionality.

ICRApplication

`ICRApplication` is the custom interface exposed by the root Application Object `CRApplication`. The methods and properties are listed in Table 6.1. They control access to the Report Server and its report files and allow Report, Server and Global Options object to be created. Errors returned from the Report Server are accessed through the `LastErrorCode` and `LastErrorString` properties. The Application Object is created from the `CRApplication` class constructor.

Methods	Custom Properties	Navigation
*ConnectToReportServer *DisconnectFromReportServer OpenReport LogOnServer LogOffServer CanClose *ParallelPrintReport *ConvertPSToPDF ClearError	*AutomaticLoadBalancing *RetrieveMode *OSAgentAddress *Server LastErrorCode LastErrorString	*ReportServers *ServerSettings Options

Table 6.1

Interface acquisition:

```
Dim app As ICRApplication, report As ICRReport
Set app = New CRApplication
```

Method and property calls:

```
app.ConnectToReportServer "butterfly.acme.com"
Set report = app.OpenReport("C:\SampleReports\Box.rpt")
if Err.Number <> 0 Then
   MsgBox Err.Description & ": " & app.LastErrorString
End If
```

ICRGlobalOptions

`ICRGlobalOptions` is the custom interface exposed by the Global Options Object `CRGlobalOptions`. The methods and properties are listed in Table 6.2. They control the settings of various global options on the Application Object. A Global Options Object is obtained from the `Options` property of the Application Object.

Methods	Custom Properties	Navigation Properties
	`MatchLogOnInfo` `MorePrintEngineErrorMessages`	`Parent` `Application`

Table 6.2

Interface acquisition:

```
Dim options As ICRGlobalOptions
Set options = app.Options
```

Property calls:

```
options.MorePrintEngineErrorMessages = False
options.MatchLogOnInfo = False
```

ICRServer

`ICRServer` is the custom interface exposed by the Configuration Server Object. The methods and properties are listed in Table 6.3. A Configuration Server Object is obtained from the `ServerSettings` property of the Application Object.

Methods	Custom Properties	Navigation Properties

	`Version`
	`HostName`
	`HostAddress`
	`InstallDir`
	`SampleReportDir`
	`SampleOutputDir`
	`WebServerDir`
	`OutputDir`
	`GatewayName`
	`DORTempDir`	

Table 6.3

The Configuration Server Object allows the client to request configuration information from the Report Server such as the Parallel Crystal Version, and the installation path names for several important directories. The Configuration Server is described in more detail in the section entitled Parallel Crystal Configuration Server.
Interface acquisition:

```
Dim config As ICRServer
Set config = app.ServerSettings
```

Property calls:

```
Dim webDir As String, reportDir As String, outputDir As String
webDir = config.WebServerDir
reportDir = config.SampleReportDir
outputDir = config.SampleOutputDir
```

ICRReport

`ICRReport` is the custom interface exposed by the Report Object. The methods and properties are listed in Table 6.4. The methods allow you to control the selection of the output format and trigger report generation. The custom properties allow report attributes to be modified. The navigation properties allow child objects to be created and the parent and application objects to be located. A Report Object is obtained by calling the `OpenReport` method of the Application Object, or by calling the `OpenSubreport` method of the Report Object.

Methods	Custom Properties	Navigation Properties
*OutputToPS *OutputToPDF *ExportToPDF *ExportToHTML *ExportToExcel *ExportToWord Export OpenSubreport DiscardSavedData Save SelectPrinter CancelPrinting PrintOut *StartParallelPrint *ConvertPSToPDF *RunScript *ViewLocalPDFFile ClearError	Title HasSavedData NumberOfGroup PrintDate GroupSelectionFormula RecordSelectionFormula SQLQueryString ProgressDialogEnabled ParameterPromptingEnabled LastErrorCode LastErrorString	ParameterFields SummaryFields FormulaFields Options ExportOptions PrinterInfo ReportSummaryInfo PrintingStatus DataBase RecordSortFields Sections Areas Parent Application PDFOutputOptions

Table 6.4

Interface acquisition:

```
Dim report1 As ICRReport, report2 As ICRReport
report1 = app.OpenReport("C:\SampleReports\Box.rpt")
reportr = report1.OpenSubreport("Studio one")
```

Method and property calls:

```
report.Title = "Sample run"
report.RecordSelectionFormula = "{Employee.Employee ID} < 10"
report.OutputToPDF "C:\SampleOutputs\Box.pdf"
```

ICRArea(s)

`ICRAreas` is the collection interface exposed by the Areas Collection Object. The methods and properties are listed in Table 6.5. An Areas Collection Object is obtained from the `Areas` property of the parent Report Object.

Each item in the collection is an Area Object representing an area within the parent report, and the number of Area Objects is returned by the `Count` property. Individual Area Objects are obtained by the `Item` method that takes an index or string argument identifying the area required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Area Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.5

`ICRArea` is the custom interface exposed by each Area Object in the collection. The methods and properties are listed in Table 6.6. They provide access to the area properties and its constituent sections.

Methods	Custom Properties	Navigation

	`Kind`
	`GroupNumber`
	`CopiesToPrint`
		`Sections`
		`Options`
		`GroupOptions`
		
		`Parent`
		`Report`
		`Application`

Table 6.6

Interface acquisition:

```
Dim ac As ICRAreas, area1 As ICRArea, area2 As Area
Set ac = report.Areas
Set area1 = ac.Item(1)
Set area2 = ac(2)
```

Collection iteration:

```
For Each area In report.Areas
   Select Case area.Kind
   Case CrReportHeader To CrGroupHeader
      ...
   Case CrDetail
      ...
   Case CrGroupFooter To CrReportFooter
      ...
   End Select
Next
```

ICRAreaOptions

`ICRAreaOptions` is the custom interface exposed by the Area Options Object. The methods and properties are listed in Table 6.7. They allow you to control details of the layout and format for the area. An Area Options Object is obtained from the `Options` property of the parent Area Object.

Methods	Custom Properties	Navigation
	KeepTogether KeepTogetherFormula NewPageAfter NewPageAfterFormula PrintAtBottomOfPage PrintAtBottomOfPageFormula ResetPageNumberAfter ResetPageNumberAferFormula Visible VisibleFormula	Parent Report Application

Table 6.7

Interface acquisition:

```
Dim options As ICRAreaOptions
Set options = area.Options
```

Property calls:

```
options.NewPageBefore = False
options.NewPageAfter = True
Set report = options.Report
```

ICRGroupAreaOptions

`ICRGroupAreaOptions` is the custom interface exposed by the Group Area Options Object. The methods and properties are listed in Table 6.8. They allow you to control the details of the layout of a group area. A Group Area Options Object is obtained from the `GroupOptions` property of the parent Area Object.

Methods	Custom Properties	Navigation

	`Condition`
	`DiscardOtherGroups`
	`KeepGroupTogether`
	`NumberOfTopOrBottomGroups`
	`RepeatGroupHeader`
	`SortDirection`
	`TopOrBottomNGroups`
		`Parent`
		`Report`
		`Application`

Table 6.8

Interface acquisition:

```
Dim options As ICRGroupAreaOptions
Set options = area.GroupOptions
```

Property calls:

```
options.SortDirection = crAscendingOrder
options.NumberOfTopOrBottomGroups = 3
options.TopOrBottomNGroups = crTopNGroups
```

ICRSection(s)

`ICRSections` is the collection interface exposed by the Sections Collection Object. The methods and properties are listed in Table 6.9. A Sections Collecction Object is obtained from the `Sections` property of the Area Object.

Each item in the collection is a Section Object representing a section within the parent area, and the number of Section Objects is returned by the `Count` property. Individual objects are obtained by the `Item` method that takes an index or string argument identifying the section required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Section Objects.

Methods	Custom Properties	Navigation
Item	Count _NewEnum	Parent Report Application

Table 6.9

ICRSection is the custom interface exposed by each Section Object in the collection. The methods and properties are listed in Table 6.10. They provide access to the section's properties and its constituent sub-report objects.

Methods	Custom Properties	Navigation
	Height Number Width	Options ReportObjects Parent Report Application

Table 6.10

Interface acquisition:

```
Dim scns As ICRSections, section As ICRSection
Set scns = area.Sections
Set section = scns.Item(2)
Set section = scns(1)
Set section = scns.Item("PH")
```

Collection iteration:

```
For Each section In area.Sections
   section.Height = 200 ' twips
   section.Width = 200  ' twips
Next
```

ICRSectionOptions

`ICRSectionOptions` is the custom interface exposed by the Section Options Object. The methods and properties are listed in Table 6.11. Section Options control the layout and formatting of an individual section within an area, whereas Area Options apply globally to all sections in an area. When conflicts occur, the True value overrides False and determines whether the property applies locally or globally. A Section Options Object is obtained from the Options property of the Section Object.

Methods	Custom Properties	Navigation Properties
	BackColor BackColorFormula FreeFromPlacement KeepTogetherFormula NewPageAfter NewPageAfterFormula NewPageBefore NewPageBeforeFormula PrintAtBottomOfPage PrintAtBottomOfPageFormula ResetPageNumberAfter ResetPageNumberAfterFormula SuppressIfBlank UnderlaySection UnderlaySectionFormula Visible VisibleFormula	Parent Report Application

Table 6.11

Interface acquisition:

```
Dim options As ICRSectionOptions
Set options = section.Options
```

Property calls:

```
options.BackColor = crBlue
options.NewPageBefore = True
options.UnderlaySection = True
```

ICRReportObject(s)

`ICRReportObjects` is the collection interface exposed by the Reports Collection Object. The methods and properties are listed in Table 6.12. A Reports Collection Object is obtained from the `ReportObjects` property of the Section Object.

Each item in the collection is a Subreport Object representing a sub-report within the parent section, and the number of Subreport Objects is returned by the `Count` property. Individual sub-reports are obtained by the `Item` method that takes an index or string argument identifying the report required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Subreport Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.12

`ICRSubReportObject` is the custom interface exposed by each Subreport Object in the collection. The methods and properties are listed in Table 6.13. They provide access to the sub-report's name and kind, and parent report, and allow a Report Object to be created for the sub-report.

Methods	Custom Properties	Navigation

	`Kind`
	`Links`
	`Name`
		`Parent`
		`Report`
		`Application`

Table 6.13

Interface acquisition:

```
Dim srptc As ICRReportObjects, srpt As ICRSubReportObject
Set srptc = section.ReportObjects
Set srpt = srptc.Item(1)
Set srpt = srptc(2)
```

Collection iteration:

```
Dim report As ICRReport, subreport As ICRReport
For Each srpt In section.ReportObjects
   report = srpt.Report
   subreport = report.OpenReport(srpt.Name)
Next
```

ICRSortField(s)

`ICRSortFields` is the collection interface exposed by the SortFields Collection Object. The methods and properties are listed in Table 6.14. A SortFields Collection Object is obtained from the `RecordSortFields` property of the Report Object.

Each item in the collection is a SortField Object representing a record sort-field within the parent report, and the number of SortField Objects is returned by the `Count` property. Individual sort fields are obtained by the `Item` method that takes an index or argument identifying the field required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of SortField Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.14

`ICRSortField` is the custom interface exposed by each SortField Object in the collection. The methods and properties are listed in Table 6.15. They provide access to the number and sort direction for the field.

Methods	Custom Properties	Navigation

	`Field`
	`SortDirection`
		`Parent`
		`Report`
		`Application`

Table 6.15

Interface acquisition:

```
Dim sfldsc As ICRSortFields, field As ICRSortField
Set sfldsc = report.RecordSortFields
Set field = sfldsc.Item(1)
Set field = sfldsc(2)
```

Collection iteration:

```
For Each field In report.RecordSortFields
   field.SortDirection = crAscendingOrder
Next
```

ICRDataBase

`ICRDataBase` is the custom interface exposed by the Database Object. The methods and properties are listed in Table 6.16. They provide access to the database tables and parameters used in the report. A DataBase Object is obtained from the Database property of the parent Report Object.

Methods	Custom Properties	Navigation Properties

`Verify`
`ConvertDatabaseDriver`
		`Parameters`
		`Tables`
		
		`Parent`
		`Report`
		`Application`

Table 6.16

Interface acquisition:

```
Dim db As ICRDataBase
Set db = report.Database
```

Methods calls:

```
db.Verify
```

ICRDataBaseTable(s)

`ICRDataBaseTables` is the collection interface exposed by the DataBaseTables Collection Object. The methods and properties are listed in Table 6.17. A databaseTables Collection Object is obtained from the `Tables` property of the Database Object.

Each item in the collection is a Database Table Object representing a table within the database, and the number of Database Table Objects is returned by the `Count` property. Individual Database Table Objects are obtained by the `Item` method that takes an index or argument identifying the table required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Database Table Objects.

Methods	Custom Properties	Navigation
Item	Count _NewEnum	Parent Report Application

Table 6.17

ICRDataBaseTable is the custom interface exposed by each Database Table Object in the collection. The methods and properties are listed in Table 6.18. They provide access to the attributes of the table and its constituent fields.

Methods	Custom Properties	Navigation
SetLogOnInfo TestConnectivity	DescriptiveName DllName Location LogOnDatabaseName LogOnUserID Name SessionUserID Type	Fields Parent Report Application

Table 6.18

Interface acquisition:

```
Dim dbtblc As ICRDataBaseTables, dbtable As ICRDataBaseTable
Set dbtblc = db.Tables
Set dbtable = dbtblc.Item(1)
Set dbtable = dbtblc(2)
```

Collection iteration:

```
For Each dbtable In db.Tables
   If Not dbtable.TestConnectivity Then
      ReportError "Cannot logon to table " & dbtable.Name
   End If
Next
```

ICRDataBaseFieldDefinition(s)

`ICRDataBaseFieldDefinitions` is the collection interface exposed by the DatabaseFields Collection Object. The methods and properties are listed in Table 6.19. A DatabaseFields Collection Object is obtained from the `Fields` property of the Database Table Object.

Each item in the collection is a DatabaseField Definition Object representing a field within the database table, and the number of DatabaseField Definition Objects is returned by the `Count` property. Individual DatabaseField Definition Objects are obtained by the `Item` method that takes an index or argument identifying the field required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of DatabaseField Definition Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.19

`ICRDataBaseFieldDefinition` is the custom interface exposed by each DatabaseField Object in the collection. The methods and properties are listed in Table 6.20. They provide access to the name, kind, and content attributes of the field.

Methods	Custom Properties	Navigation

	`Kind`
	`Name`
	`Text`
		`Parent`
		`Report`
		`Application`

Table 6.20

Interface acquisition:

```
Dim dbfldc As ICRDataBaseFieldDefinitions
Dim field As ICRDatabaseFieldDefinition
Set dbfldc = table.Fields
Set field = dbfldc.Item(1)
Set field = dbfldc(2)
```

Collection iteration:

```
For Each field In table.Fields
   If field.Kind = crFormulaField Then
      Dim fname As String
      fname = field.Name
   End If
Next
```

ICRDataBaseParameter(s)

`ICRDataBaseFieldparameters` is the collection interface exposed by the DatabaseParameters Collection Object. The methods and properties are listed in Table 6.21. A DatabaseParameters Collection Object is obtained from the `Parameters` property of the Database Object.

Each item in the collection is a Database Parameter Object representing a stored procedure or parameterized query within the database, and the number of Database Parameter Objects is returned by the `Count` property. Individual Database Parameter Objects are obtained by the `Item` method that takes an index or argument identifying the parameter required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Database Parameter Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.21

`ICRDataBaseParameter` is the custom interface exposed by each Database Parameter Object in the collection. The methods and properties are listed in Table 6.22. They provide access to the name, type and value attributes of the parameter.

Methods	Custom Properties	Navigation
	Name Type Value	Parent Report Application

Table 6.22

Interface acquisition:

```
Dim dbpmc As ICRDataBaseParameters
Dim param As ICRDataBaseParameter
Set dbpmc = dbase.Parameters
Set param = dbpmc.Item(1)
Set param = dbpmc(2)
```

Iteration collection:

```
For Each param in dbase.Parameters
   If param.Type = crDateField Then
      Dim val As String
      val = param.Value
   End If
Next
```

ICRPrintingStatus

`ICRPrintingStatus` is the custom interface exposed by the PrintingStatus Object. The methods and properties are listed in Table 6.23. They provide access to the number of records and pages printed, and the overall print status for the parent report. A PrintingStatus Object is obtained from the `PrintingStatus` property of the parent report Object.

Methods	Custom Properties	Navigation
	LatestPageNumber NumberOfPages Progress RecordsPrinted RecordsSelected	Parent Report Application

Table 6.23

Interface acquisition:

```
Dim pstats As ICRPrintingStatus
Set pstats = report.PrintingStatus
```

Method and Property Calls:

```
Dim pages As Integer, records As Integer, selected As Integer
pages = pstats.NumberOfPages
records = pstats.ReordsPrinted
selected = pstats.RecordsSelected
```

ICRPrinterInfo

`ICRPrinterInfo` is the custom interface exposed by the PrinterInfo Object. The methods and properties are listed in Table 6.24. They provide access to the selected printer's port, driver and name. A PrinterInfo Object is obtained from the `PrinterInfo` property of the parent Report Object.

Methods	Custom Properties	Navigation Properties

	`DriverName`
	`PortName`
	`PrinterName`
		`Parent`
		`Report`
		`Application`

Table 6.24

Interface acquisition:

```
Dim printer As ICRPrinterInfo
Set printer = report.PrinterInfo
```

Method and property calls:

```
Dim info As String
info = "Printer: " & printer.PrinterName & " " & _
       "Driver: " & printer.DriverName & " " & _
       "Port: " & printer.Port
```

ICRReportSummaryInfo

`ICRReportSummaryInfo` is the custom interface exposed by the ReportSummary Object. The methods and properties are listed in Table 6.25. They provide access to summary information for the parent Report Object such as author, subject and title. A Report Summary Object is obtained from the `ReportSummaryInfo` property of the parent Report Object.

Methods	Custom Properties	Navigation Properties
	Author Comments Keywords Name Subject Template Title	Parent Report Application

Table 6.25

Interface acquisition:

```
Dim info As ICRReportSummaryInfo
Set info = report.ReportSummaryInfo
```

Method and property calls:

```
info.Title = "Sample Monthly Lists"
info.Comments = "Just a demonstration"
```

ICRExportOptions

`ICRExportOptions` is the custom interface exposed by the ExportOptions Object. The methods and properties are listed in Table 6.26. They allow the output format and destination of the parent report to be specified. An ExportOptions Object is obtained from the `ExportOptions` property of the parent Report Object.

Methods	Custom Properties	Navigation Properties

`Reset`	`CharFieldDelimiter`
	`CharStringDelimier`
	`DestinationDllName`
	`DestinationType`
	`DiskFileName`
	`ExchangeFolderPath`
	`ExchangeDestination`
	`ExchangePassword`
	`ExchangeProfile`
	`FormatDllName`
	`FormatType`
	`HTMLFileName`
	`MailBccList`
	`MailMessage`
	`MailSubject`
	`MailToList`
	`MailCcList`
	`NumberOfLinesPerPage`
	`ODBCExportTableName`
	`ODBCDataSourcePassword`
	`ODBCDataSourceUserID`
	`ODBCDataSourceName`
	`UseReportNumberFormat`
	`UseReportDateFormat`
		`Parent`
		`Report`
		`Application`

Table 6.26

You must set both `FormatType` and `DestinationType`, and then depending upon their values, set additional properties appropriately. The values of `FormatType` are supplied by a set of constants of the form `crEFTXXX` listed in the online documentation, and the values of `DestinationType` are supplied by a corresponding set of constants of the form `crEDTXXX`. When all properties are set, you must call the `Export` method of the Report Object to register the settings for the current report. When you call the `PrintOut` method, the report is generated according to the export settings.

Interface acquisition:

```
Dim options As ICRExportOptions
options = report.ExportOptions
```

Method and property calls:

```
' Generate report in Excel format and store on disk
options.FormatType = crEFTExcel40
options.DestinationType = crEDTDiskFile
options.DiskFileName = "C:\SampleOutputs\Box.xls"
report.Export
report.PrintOut

' Generate report in Text format and send as Email
options.FormatType = crEFTText
options.DestinationType = crEDTMailMAPI
options.MailToList = "toby@acme.com"
options.MailCcList = ""
options.MailSubject = "Box Office Report"
options.MailMessage = "Here is your Box Office Report" & _
                      Chr(13) & Chr(10)
report.Export
report.PrintOut
```

ICRReportOptions

`ICRReportOptions` is the custom interface exposed by the ReportOptions Object. The methods and properties are listed in Table 6.27. They provide access to a variety of option settings that may be set for the current report. A ReportOptions Object is obtained from the `Options` property of the parent Report Object.

Methods	Custom Properties	Navigation Properties

	`CaseInsensitiveSQLData`
	`ConvertDateTimeType`
	`ConvertNullFieldToDefault`
	`MorePrintEngineErrorMessages`
	`SaveDataWithReport`
	`SaveSummariesWithReport`
	`TranslateDosMemos`
	`TranslateDosStrings`
	`UseIndexForSpeed`
	`VerifyOnEveryPrint`
		`Parent`
		`Report`
		`Application`

Table 6.27

Interface acquisition:

```
Dim options As ICRReportOptions
Set options = report.Options
```

Method and property calls:

```
options.VerifyOnEveryPrint = False
options.UseIndexForSpeed = True
```

ICRFormulaFieldDefinition(s)

`ICRFormulaFieldDefinitions` is the collection interface exposed by the FormulaFieldDefinitions Collection Object. The methods and properties are listed in Table 6.28 on page 208. A FormulaFieldDefinitions Collection Object is obtained from the `FormulaFields` property of the Report Object.

Each item in the collection is a FormulaField Definition Object representing information stored in a formula field in the report, and the number of FormulaField Definition Objects is returned by the `Count` property. Individual FormulaField Definition Objects are obtained by the `Item` method that takes an index or argument identifying the parameter required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Database Parameter Objects.

Methods	Custom Properties	Navigation

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.28

`ICRFormulaFieldDefinition` is the custom interface exposed by each FormulaField Definition Object in the collection. The methods and properties are listed in Table 6.29. They provide access to the name, kind, and content attributes of the field.

Methods	Custom Properties	Navigation Properties

`Check`	`Kind`
	`FormulaFieldName`
	`Name`
	`Text`
		`Parent`
		`Report`
		`Application`

Table 6.29

Interface acquisition:

```
Dim ffldc As ICRFormulaFieldDefinitions
Dim field As ICRFormulaFieldDefinition
Set ffldc = report.FormulaFields
Set field = ffldc.Item(1)
Set field = ffldc(2)
```

Collection iteration:

```
For Each field In report.FormulaFields
   If field.Kind = crParameterField Then
      field.Text = field.Text & "*10"
      If Not field.Check Then Exit For
   End If
Next
```

ICRParameterFieldDefinition(s)

`ICRParameterFieldDefinitions` is the collection interface exposed by the ParameterFieldDefinitions Collection Object. The methods and properties are listed in Table 6.30. A ParameterFieldDefinitions Collection Object is obtained from the `ParameterFields` property of the Report Object.

Each item in the collection is a ParameterField Definition Object representing information stored in a parameter field in the report, and the number of ParameterField Definition Objects is returned by the `Count` property. Individual ParameterField Definition Objects are obtained by the `Item` method that takes an index or argument identifying the parameter required. Visual Basic clients may iterate through all items with the `For Each` statement that calls the `_NewEnum` property to create an enumeration of Database Parameter Objects.

Methods	Custom Properties	Navigation Properties

`Item`	`Count`
	`_NewEnum`
		`Parent`
		`Report`
		`Application`

Table 6.30

`ICRParameterFieldDefinition` is the custom interface exposed by each ParameterField Definition Object in the collection. The methods and properties are listed in Table 6.31. They provide access to the name, kind, and content attributes of the field.

Methods	Custom Properties	Navigation Properties
SetCurrentValue SetDefaultValue	CurrentValue CurrentValueSet DefaultValueSet Kind Name NeedsCurrentValue Prompt ParameterFieldName ReportName ValueType	Parent Report Application

Table 6.31

Interface acquisition:

```
Dim pfldc As ICRParameterFieldDefinitions
Dim param As ICRparameterFieldDefinition
Set pfldc = report.ParameterFields
Set param = pfldc.Item(1)
Set param = pfldc(2)
```

Collection iteration:

```
Dim pval As Variant
For Each param In report.ParameterFields
   If param.NeedsCurrentValue Then
      param.SetCurrentValue pval
   Else
      pval = param.CurrentValue
   End If
Next
```

PDF Report Generation

Parallel Crystal Report Servers can generate reports in PDF using "direct" and "indirect" formatting schemes. The direct scheme uses the Dynalivery proprietary PDF Generation Library that generates PDF directly from the internal representation held in the report file. The indirect scheme generates your report in a temporary PostScript file, and then converts the file to PDF using a utility from Adobe called Acrobat Distiller. The direct scheme is faster and is the recommended default. Table 6.32 shows how these schemes are used by the various interfaces and methods in the Automation Server.

Interface	Method	PDF Generation Scheme
ICRReport	OutputToPDF ExportToPDF ConvertPSToPDF	Dynalivery PDF Library Acrobat Distiller Acrobat Distiller

Table 6.32

Direct PDF generation using `OutputToPDF` has already been described in the previous Chapter. The equivalent indirect method `ExportToPDF` is used in the same way. For example:

```
' Visual Basic
Dim app As ICRApplication, report As ICRReport
...
report = app.OpenReport("C:\SampleReports\Box.rpt")
...
' Direct generation with Dynalivery PDF Library
report.OutputToPDF "C:\SampleOutputs\Box1.pdf"
...
' Indirect generation with Acrobat Distiller
report.ExportToPDF "C:\SampleOutputs\Box2.pdf"
```

Although `OutputToPDF` is the faster of the two methods, `ExportToPDF` may be preferable if your report is image rich. In general, we recommend that you try `OutputToPDF` first of all, and then use `ExportToPDF` if the results are not satisfactory.

Parallel Crystal Configuration Server

When Parallel Crystal is installed, a number of directories are created with sample Crystal Reports and demonstration materials. In order to help you locate path names for these directories, Parallel Crystal incorporates a Configuration Server that returns configuration- and installation-dependent information to the client.

Automation clients access the Configuration Server through the properties of the `ICRServer` interface that is returned by the `ServerSettings` method of the `ICRApplication` interface. `ServerSettings` is declared as follows:

```
// Visual C++
HRESULT get_ServerSettings( LPDISPATCH *pConfigServer )

' Visual Basic
Property Get ServerSettings As Object

// Visual J++
public java.lang.Object getServerSettings()
```

Typically, you get a reference to the `ICRServer` interface as follows:

```
// Visual C++
ICRServer *config;
config = static_cast<ICRServer *>(app->ServerSettings);

' Visual Basic
Dim config As ICRServer
Set config = app.ServerSettings

' VBScript
Dim config
Set config = app.ServerSettings

// Visual Java
ICRServer config;
config = (ICRServer)app.getServerSettings()
```

The `ICRServer` interface contains a series of read-only properties that return strings containing configuration-dependent information about the Report Server. These properties are declared as follows:

```
// Visual C++
HRESULT get_Version( BSTR *pDirPath )
HRESULT get_HostName( BSTR *pDirPath )
HRESULT get_HostAddress( BSTR *pDirPath )
HRESULT get_InstallDir( BSTR *pDirPath )
HRESULT get_SampleReportDir( BSTR *pDirPath )
HRESULT get_SampleOutputDir( BSTR *pDirPath )
HRESULT get_WebServerDir( BSTR *pDirPath )

' Visual Basic
Property Get Version As String
Property Get HostName As String
Property Get HostAddress As String
Property Get InstallDir As String
Property Get SampleReportDir As String
Property Get SampleOutputDir As String
Property Get WebServerDir As String

// Visual J++
java.lang.String getVersion()
java.lang.String getHostName()
java.lang.String getHostAddress()
java.lang.String getInstallDir()
java.lang.String getSampleReportDir()
java.lang.String getSampleOutputDir()
java.lang.String getWebServerDir()
```

Where:

`Version`	Returns the Parallel Crystal version string.
`HostName`	Returns the Parallel Crystal Report Server host name.
`HostAddress`	Returns the Parallel Crystal Report Server host address.
`InstallDir`	Returns the Parallel Crystal installation root directory.
`SampleReportDir`	Returns the Parallel Crystal sample reports directory.
`SampleOutputDir`	Returns the Parallel Crystal sample outputs directory.
`WebServerDir`	Returns the Web Server's root directory.

In the following example, the procedure `RunSample` takes the name of a Report Server host and a report file in `SampleReportDir` and retrieves the generated HTML report to the `WebServerDir`. For example, if we call RunSample with:

```
RunSample("flea.acme.com", "FTE")
```

it will connect to the ReportServer on `"flea.acme.com"`, run the report `"FTE.rpt"`, export the result to `"FTE.html"`, and retrieve the HTML file back to Web Server's root directory[1]. `RunSample` is shown below using Visual C++ Visual Basic and Visual J++

[1] We'll discuss report retrieval in more detail in the next section.

interfaces to the Automation Server. In each case, the caller must be able to handle exceptions raised when the Automation Server methods return a COM error.

```
// Visual C++: caller must handle _com_error's.
void RunSample( _bstr_t host, _bstr_t report )
{
   ICRApplicationPtr pApp(__uuidof(CRApplication));
   pApp->ConnectToReportServer(host);
   ICRServer *pConfig =
      static_cast<ICRServer *>(pApp->ServerSettings);
   _bstr_t inputPath = pConfig->SampleReportDir + report + ".rpt";
   _bstr_t outputPath = pConfig->WebServerDir + report + ".html";
   pApp->RetrieveMode = TRUE;
   ICRReport *pReport =
      static_cast<ICRReport *>(pApp->OpenReport(inputPath));
   pReport->ExportToHTML(outputPath);
   pConfig->Release();     // Must release ICRServer!
   pReport->Release();     // Must release ICRReport!
   pApp->DisconnectFromReportServer();
   pApp->Release();        // Must release ICRApplication!
}

' Visual Basic: caller must have an On Error handler
Sub RunSample( ByVal host As String, ByVal report As String )
   Dim pApp As ICRApplication, pReport As ICRReport
   Dim pConfig As ICRServer
   Dim inputPath As String, outputPath As String
   Set pApp = New CRApplication
   pApp.ConnectToReportServer(host)
   Set pConfig = pApp.ServerSettings
   inputPath = pConfig.SampleReportDir & report & ".rpt"
   outputPath = pConfig.WebServerDir & report & ".html"
   pApp.RetrieveMode = True
   Set pReport = pApp.OpenReport(inputPath)
   pReport.ExportToHTML(outputPath)
   pApp.DisconnectFromReportServer()
End Sub

// Visual J++: caller must handle COMFailExceptions
void RunSample( String host, String report )
{
   String inputPath, outputPath;
   ICRApplication pApp = (ICRApplication)new CRApplication();
   pApp.ConnectToReportServer(host);
   ICRServer pConfig = (ICRServer)pApp.getServerSettings();
   inputPath = pConfig.getSampleReportDir() + report + ".rpt";
   outputPath = pConfig.getWebServerDir() + report + ".html";
   pApp.putRetrieveMode(1);
   Variant v = new Variant(); v.noParam();
   ICRReport pReport = (ICRReport)pApp.OpenReport(inputPath, v);
   pReport.ExportToHTML(outputPath);
   pApp.DisconnectFromReportServer();
}
```

Parallel Crystal Report Retrieval

When a report is generated using any of the file-based formats, the file is created on the Report Server. If the client and the Report Server do not run on the same machine, then the report file must be retrieved if the client wishes to view it locally. The easiest way to do this is to use the Parallel Crystal Report Retrieval Service.

When report retrieval is enabled, the report is copied back to a named file on the client machine. If you have a local viewer on your client such as a web browser or a PDF Reader, then you can display the report as soon as it is returned. If your client is a VBScript application running on the IIS Web Server, then you can return a hyperlink to the report so that the Web Browser can finally display it.

Report retrieval is enabled for automation clients by setting the `RetrieveMode` property of the `ICRApplication` interface to `True`. You must do this after you have called `ConnectToReportServer`. Then when you set the output format with a call such as `ExportToHTML` or `OutputToPDF`, you must specify a path name for the file on the *client's* machine rather than the Report Server machine.

```
// Visual C++
ICRApplicationPtr pApp(__uuidof(CRApplication));           // 1.
pApp->ConnectToReportServer("example.host.com");           // 2.

pApp->RetrieveMode = TRUE;                                 // 3.

// C:\SampleReports\\Box.rpt on the Report Server machine.
ICRReport *pReport =                                       // 4.
   static_cast<ICRReport *>
      (pApp->OpenReport("C:\\SampleReports\\Box.rpt"));
// C:\MyReports\Box.pdf on the client machine.
pReport->OutputToPDF("C:\\MyReports\\Box.pdf");            // 5.

pReport->Release();       // Must release ICRReport!
pApp->DisconnectFromReportServer();                        // 6.
pApp->Release();          // Must release ICRApplication!



' Visual Basic
Dim pApp As ICRApplication, pReport As ICRReport
Set pApp = New CRApplication                               ' 1.
pApp.ConnectToReportServer("example.host.com")             ' 2.

pApp.RetrieveMode = True                                   ' 3.

' C:\SampleReports\Box.rpt on the Report Server machine.
Set pReport = pApp.OpenReport("C:\\SampleReports\\Box.rpt") ' 4.
' C:\MyReports\Box.pdf on the client machine.
pReport.OutputToPDF("C:\\MyReports\\Box.pdf")              ' 5.

pApp.DisconnectFromReportServer()                          ' 6.
```

```
// Visual J++
ICRApplication pApp = (ICRApplication)new CRApplication();  // 1.
pApp.ConnectToReportServer("example.host.com");
       // 2.

pApp.putRetrieveMode(1);                                    // 3.

Variant v = new Variant(); v.noParam();
// C:\SampleReports\Box.rpt on the Report Server machine.
ICRReport pReport = (ICRReport)
   pApp.OpenReport("C:\\SampleReports\\Box.rpt", v);        // 4.
// C:\MyReports\Box.pdf on the client machine.
pReport.OutputToPDF("C:\\MyReports\\Box.pdf");              // 5.

pApp.DisconnectFromReportServer();                          // 6.
```

Each numbered paragraph below corresponds to the same numbered code statement

1. An Application Object is created and a reference to the `ICRApplication` interface is stored in `pApp`.

2. The client connects to the Report Server running on the host `"example.host.com"`.

3. The client sets the `RetrieveMode` property to `TRUE`, to enable the retrieval service.

4. A Report Object is opened using the path name of the `Box` report on the Report Server machine. The refererence to the `ICRReport` interface is stored in `pReport`.

5. The `OutputToPDF` method is called to generate the report in PDF. The argument specifies the path name of the PDF file for the client machine. The method returns when the report has been retrieved to the client machine.

6. The client is disconnected from the Report Server.

Automatic Report Retrieval requires your Parallel Crystal to be installed with a file transfer service. If this service is not running, then the call to `OutputToPDF` will fail. If you call the `LastErrorString` property of `ICRReport`, it will return the message:

PCREServerError: Connect: cannot connect to PCRE File Transfer Service

One you have connected to a Report Server, you can switch the Report Retrieval on or off, by setting the `ReportRetrieval` property to `True` or `False`.

Parallel Crystal Load Balancer

When your client has access to a group of Report Servers, you can use the Parallel Crystal Load Balancer service to ensure that the report processing load is distributed appropriately across the available server machines. The load balancer provides two interfaces for automatic and manual use.

When you use automatic mode, the client is connected to a Report Server chosen by the Load Balancer. The choice is made by a selection algorithm that is normally configured by the System Administrator. The two algorithms most commonly used are called "load factor" and "round robin". The "load factor" algorithm compares all the Report Servers in the group and returns the one with the smallest load. The "round robin" algorithm simply cycles through all Report Servers in turn and selects the next one in the sequence.

When you use manual mode, the Load Balancer returns a collection of Report Server hosts together with their load factors. You can iterate through the collection and choose a server based on your own selection criteria. Then you connect by passing the server to the Application Object.

You should bear in mind that when you use the Load Balancer in Automatic Mode, you cannot tell in advance which Report Server your client will connect to. Therefore, you must ensure that your report files are accessible from all possible servers in the group. This can be achieved either by having shared or mirrored disk storage.

Using the Load Balancer in Automatic Mode

Using the Load Balancer in automatic mode is very simple. First, you set the Application Object's `AutomaticLoadSharing` property to `True`, and then you call `ConnectToReportServer` without specifying a Report Server host name.

```
// Visual C++
ICRApplicationPtr pApp(__uuidof(CRApplication));
pApp->AutomaticLoadSharing = TRUE;
pApp->ConnectToReportServer("");

// Visual Basic
Dim pApp As ICRApplication
Set pApp = New CRApplication
pApp.AutomaticLoadSharing = True
pApp.ConnectToReportServer ""

// VBScript
Dim pApp
Set pApp = Server.CreateObject("MAS.CRApplication.1")
pApp.AutomaticLoadSharing = True
pApp.ConnectToReportServer ""

// Visual J++
ICRApplication pApp = (ICRApplication)new CRApplication();
pApp.putAutomaticLoadSharing(true);
pApp.ConnectToReportServer("");
```

Note that the AutomaticLoadSharing property must be set before the call to ConnectToReportServer. If you forget to set the property, then each of the calls to ConnectToReportServer will locate a Report Server somewhere on your network, without involving the Load Balancer.

If you do supply a host name when AutomaticLoadSharing is true, then ConnectToReportServer will try to use the Load Balancer on the specified host. However, if there is no Load Balancer running on the host, ConnectToReportServer will fail, and LastErrorString will return the message:

```
"Server error: cannot connect client to Load Balancer Service"
```

In general, it's simplest just to use the automatic mode as shown.

If the Load Balancer is unable to locate any Report Servers, calls to ConnectToReportServer will fail and LastErrorString will return the message:

```
"Server error: Load Balancer failure: no available Report Servers"
```

This error can occur either because no Report Servers were running, or because the installed Report Server have not been configured to use the Load Balancer Service. You should check with installation and configuration of Parallel Crystal if this error persists.

Using the Load Balancer in Manual Mode

When you use manual mode, the Load Balancer returns a list of available Report Servers. You apply your own selection criteria to the list and then call the ConnectToSelectedReportServer to make the connection.

The list of Report Servers is accessed through a ReportServers Collection Object whose interface is called ICRReportServers. The methods and properties of this interface are shown in Table 6.33. You obtain a ReportServers Collection Object from the ReportServers property of the Application Object. Each item in the collection contains information about a particular Report Server, and the number of available Report Servers is returned by the Count property. Individual Report Servers are obtained by the Item method that takes an index or argument identifying the parameter required. Visual Basic clients may iterate through all items with the For Each statement that calls the _NewEnum property to create an enumeration of ReportServer Objects.

Methods	Custom Properties	Navigation Properties
Item	Count _NewEnum	Parent Report Application

Table 6.33

The details of each Report Server Object are available through the `ICRReportServer` interface whose methods and properties are listed in Table 6.34.

Methods	Custom Properties	Navigation Properties

	`Loadfactor`
	`NumberOfReportServerProcessors`
	`NumberOfReportEngines`
	`ReportServerAddress`	

Table 6.34

- `ReportServerAddress` is the IP address for the Report Server host. If you select this Report Server, your client will connect to this address.
- `NumberOfReportEngines` is the number of Report Engine Server processes currently running on the Report Server.
- `NumberOfReportServerProcessors` is the number of CPU processors on the Report Server host.
- `Loadfactor` is a measure of the reporting load on the Report Server. It is currently given by

  ```
  NumberOfReportEngines/NumberOfReportServerProcessors
  ```

  If you have 4 Report Engine Server processes on a 2 processor machine, then the load factor is 2. If you have 4 Report Engine Server processes on a 1 processor machine, then the load factor is 4.

In the following sections we demonstrate how to call the LoadBalancer manually using Visual C++, Visual Basic and Visual J++. The example retrieves the list of available Report Servers from the `ReportServers` property, and then iterates through the list, locating the server with the smallest load factor. The client connects to this server by passing it to the Application Object's `ConnectToSelectedReportServer` method. For brevity and simplicity, details of error reporting are omitted.

When using the Load Balancer in manual mode, you should bear in mind that the load factor in each of the Report Server Objects is correct when the call to the `ReportServers` property returns. However, it will become invalid as other clients connect independently to the same group of servers, and therefore it is not possible to guarantee that a choice made solely on the basis of load factors is necessarily correct. The manual interface to the Load Balancer is intended to allow Report Server selection to be based on application-dependent information used in conjunction with, or instead of, simple load factors.

Using Manual Load Balancing with Visual C++

In this example we show how to use the Load Balancer in manual mode using Visual C++ with COM support. Client connection is performed by the method `Connect` of some application class `CMyApp`. For simplicity, we assume that the Application Object reference is held in a member variable `m_pApp`.

```
void CMyApp::Connect()
{
   ICRReportServers *pSvrList =
      static_cast<ICReportServers *>(m_pApp->ReportServers);// 1.
   int nServers = pSvrList->Count;                          // 2.
   ICRReportServer *pSelected = NULL;                       // 3.
   double minLoad = 1000;

   // Iterate through the list of servers.
   for ( int i = 1; i <= nServers; i++ ) {                  // 4.
      ICRReportServer *svr =
         static_cast<ICRReportServer *>(pSvrList->Item[i]); // 5.
      double svrLoad = svr->Loadfactor;
      if ( svrLoad < minLoad ) {                            // 6.
         minLoad = svrLoad;
         if ( pSelected != NULL ) pSelected->Release();
         pSelected = svr;
      } else {
         svr->Release();
      }
   }

   // If server selected connect to it.
   if ( pSelected != NULL ) {                               // 7.
      m_pApp->ConnectToSelectedReportServer(pSelected);
      MsgBox("Connected to Report Server on host " +
             pSelected->ReportServerAddress);
   } else {
      MsgBox("No Report Servers available");
   }

   // Release interfaces.                                   // 8.
   If ( pSelected != NULL ) pSelected->Release();
   pSvrList->Release();
}
```

Each numbered paragraph below corresponds to the same numbered code statement

1. The list of available Report Servers is returned by a call to the `ReportServers` property of the Application Object.

2. The number of servers in the list is returned by the `Count` property.

3. The variable `pSelected` is used to store the identity of the Server that is selected from the list. The variable `minLoad` is used to keep a running record of the minimum server loading.

4. The loop traverses the list of servers and examines their loads one by one. The list must be indexed from 1 through the value returned by the `Count` property.

5. The `Item` property returns an `IDispatch` interface to the i'th Report Server Object in the list. A static_cast is used to simplify the conversion to the `ICRReportServer` custom interface.

6. `pSelected` is updated whenever a Server is found whose load is currently less than the value in `minLoad`. We have to be careful here and elsewhere to call `Release` on interfaces when we are finished with them. If these calls are omitted, the Automation Server will be unable to reclaim memory resources.

7. If a server has been selected, we pass it in the call to `ConnectToSelectedReportServer`. After the connection is made, we call `Release` on `pSelected` since we are finished with the interface. If no Report Server was selected, an error could be reported at this point.

8. The `ICRReportServers` and `ICRReportServer` interfaces are released to allow storage reclamation in the Automation Server.

Much of the complexity in the example derives from the need to convert from `IDispatch` interfaces to custom interfaces, and from the ever-present need to manage object lifetimes by releasing their interfaces.

Using Manual Load Balancing with Visual Basic

In this example, we show how to use the Load Balancer in manual mode using Visual Basic. We assume that `app` is a global reference to the Application Object that has been established when the main form is loaded. For simplicity, the code is presented as a single procedure that is executed when a Connect button is pressed.

```
Sub CmdConnect_Click()
   Dim selected As ICRReportServer, Server As ICRReportServer ' 1.
   Dim count As Integer, minLoad As Double
   minLoad = 1000                                             ' 2.
   selected = Nothing

   'Iterate through the collection
   For Each server In app.ReportServers                       ' 3.
      Dim load As Double
      load = server.Loadfactor                                ' 4.
      If load < minLoad Then
         minLoad = load
         selected = server
      End If
   Next

   ' Connect to selected Report Server
   If IsObject(selected) Then                                 ' 5.
      app.ConnectToSelectedReportServer selected
      MsgBox "Connectd to Report Server at host " & _         ' 6.
             selected.ReportServerAddress
   Else
      MsgBox "No Report Servers available"
   End If
End Sub
```

Each numbered paragraph below corresponds to the same numbered code statement

1. `Selected` holds a reference to an `ICRReportServer` interface and identifies the most lightly loaded Report Server. `minLoad` holds the load factor for the currently selected server.

2. The variables are initialized to represent a state in which no server has been selected.

3. The loop iterates through the collection returned by the `ReportServers` property of the Application Object.

4. The load factor for the current server in the collection is compared with `minLoad` and if it is smaller, `minLoad` and `selected` are updated.

5. If `selected` references an object in the server's collection, it is passed as an argument to the `ConnectToSelectedReportServer` method that then makes the client connection.

6. If the client is connected to the selected Report Server, a message box is displayed identifying the server's host address.

The above example can easily be adapted for VBScript by deleting the "As" clauses from the `Dim` statements.

Using Manual Load Balancing with Visual J++

In this example, we show how to use the Load Balancer in manual mode using Visual J++. For simplicity, we assume that the class member `app` holds a reference to the Application Object, and that the method `Connect` selects an appropriate Report Server.

```
public void Connect()
{
   ICRReportServer selected = null;                          // 1.
   double minLoad = 1000;
   ICRReportServers svrList =                                // 2.
      (ICRReportServers)app.getReportServers ();
   int nsvrs = svrList.getCount();                           // 3.

   // Iterate through Report Server List.
   for ( int i = 1; i <= nsvrs; i++ ) {                      // 4.
      ICRReportServer server =
         (ICRReportServer)svrList.getItem(i);                // 5.
      double load = server.getLoadfactor();                  // 6.
      if ( load < minLoad ) {
         minLoad = load;
         selected = server;
      }
   }

   // Connect to selected server.
   if ( selected != null ) {                                 // 7.
      app.ConnectToSelectedReportServer(selected);
      MsgBox("Connected to Report Server on host " +         // 8.
             selected.getHostAddress());
   } else {
      MsgBox("No Report Servers available");
   }
}
```

Each numbered paragraph below corresponds to the same numbered code statement

1. The variable `selected` holds a reference to the most lightly loaded Report Server, and the variable `minLoad` holds the load factor for selected. They are initialized to represent a state in which no server is selected.

2. The list of available Report Server is returned by the `getReportServers` property method. Note that the property returns an `Object` reference that must be cast to the custom interface type `ICRReportServers`.

3. The number of Report Servers in the list is returned by the `getCount` property method.

4. The loop indexes through the list of Report Servers. The index values run from 1 through to the value returned by `getCount`.

5. The property method `getItem` returns a reference to the i'th Report Server Object in the list. The property returns an Object reference that must be cast to the custom interface type `ICRServerLoad`.

6. The load factor for the i'th Report Server is returned by the `getLoadfactor` property method. If the value is less than `minLoad`, the values of `minLoad` and `selected` are updated.

7. If a Report Server was chosen, the value of selected is passed to the `ConnectToSelectedReportServer` method which makes the client connection.

8. If the client is successfully connected to the Report Server, a message box displays the server's host address.

Chapter 7
Trouble Shooting ActiveX Clients

This chapter describes how to solve the problems that most frequently occur with ActiveX automation clients.

Problems with Client Connections

An attempt to connect your client to the Automation Server can fail for the following reasons:

1. The Automation Server DLL `cpemas1.dll` has not been installed on the client machine. You can check for this by any of the following means:

    - Use the `regedit` tool to check the registry entries in the hive `HKEY_CLASSES_ROOT`. With the exception of the DLL path name which is installation dependent, they should be the same as those given in the section Connecting to the Automation Server.

    - Use the Visual Basic Object references dialog to locate an entry for the `cpemas1.dll`.

    - Use the Visual J++ Type Library Wizard to locate an entry for `cpemas1.dll`.

    - Use the Visual C++ OLE/COM Object Viewer and use the Type Library folder to locate an entry for `cpemas1.dll`.

2. The registry entries under `HKEY_CLASSES_ROOT` have become corrupted.

3. You have moved the `cpemas1.dll` DLL so that the path name is no longer valid.

To fix these problems, you should uninstall the Automation Server with the command

```
regsvr32.exe /u cpemas1.dll
```

and then re-check the registry to ensure the entries have been completely purged. Then re-install the Automation Server with the command

```
regsvr32.exe cpemas1.dll
```

and check the registry entries again.

When the Automation Server is correctly installed and you can get a reference to the root application interface `ICRApplication`, calls to `ConnectToReportServer` can fail for the following reasons:

4. The argument string does not specify a valid host IP address or domain name.

5. The argument string is valid, but Parallel Crystal is either not installed or not running on the host you specified.

6. If the `LastErrorString` property returns the message

   ```
   "PCRE Gateway error: cannot execute PCRE Server"
   ```

   the PCRE Gateway was unable to start a Report Engine Server process for your client. The most likely reason is that the path configured for the server's executable at installation has become invalid. You should check the Parallel Crystal installation with your System Administrator or Dynalivery Technical Support Staff.

7. If the `LastErrorString` property returns the message

   ```
   "PCRE Gateway error: timeout after 15 secs: PCRE Server not
    started"
   ```

   the PCRE Gateway was not able to start a report Engine Server process within the allotted 15 second timeout period. This occurs when the Report Server is particularly busy. If the problem persists, you should get your System Administrator to re-configure the Gateway with an extended timeout period.

8. You tried to connect using the Load Balancer but it was unable to locate any Report Servers. In this case, it is possible that the Report Servers were not configured to use the Load Balancer Service, or there is an inconsistency in the way the Load Balancer is configured. You should get your System Administrator to check the current Load Balancer configuration.

If problems connecting to the Report Server persist, the causes may be more obscure. First, you should check the status of the OSAgent that administers client/server connections by keeping a dynamic record of the CORBA servers running on your network. The following problems can arise with OSAgents:

9. If there are no OSAgents running on your network, clients will be unable to connect to any of the available Report Servers. You may find that your client hangs for around 15 seconds before `ConnectToReportServer` returns with a general failure message. You should ensure that at least one of the Report Servers on your network is running an OSAgent.

10. Your client communicates with the OSAgent using UDP broadcast packets that are normally confined to a subnet of your LAN. If you are trying to connect to a Report Server on a remote network, you may need to supply the client with the IP address of the remote OSAgent. In order to do this, you must set the `OSAgentAddress` property before you call `ConnectToReportServer`.

For example:

```
// Visual C++
app->OSAgentAddress = "123.456.654.321";
app->ConnectToReportServer("123.456.654.321");

' Visual Basic
app.OSAgentAddress = "123.456.654.321"
app.ConnectToReportServer "example.host.com"

// Visual J++
app.putOSAgentAddress("123.456.654.321");
app.ConnectToReportServer("example.host.com");
```

The IP address supplied to the OSAgent, and the host name supplied to `ConnectToReportServer` need not specify the same machine. You should get your System Administrator to supply you with the correct IP address for the Report Server running the OSAgent.

Finally, the following connection problem can occur on networks with badly configured Domain Name Servers:

11. You may find that calls to `ConnectToReportServer` succeed when the host is specified with an IP address, but fail when the host is specified with a domain name. You should check with your System Administrator that the DNS servers on all machines are configured correctly, and can handle both forward and reverse lookups.

Problems with Export Formats

In Chapter 7, the section entitled ICRExportOptions describes methods and properties for controlling the output format and destination of the report. Two problems can arise when specifying the export options:

1. Not all of the formatting capability provided through `ICRExportOptions` may be available on the Report Server. In practice, the Report Engine Server relies on a set of external DLLs provided with your Parallel Crystal Installation to provide reports generated in formats such as Word, Excel, and Lotus. If you encounter problems in generating reports in anything other than PDF, PostScript or HTML, check with your System Administrator that the DLL is available on the Report Server, and accessible through the PATH environment variable. The names of the formatting DLLs are listed in the section Print Job Outputs in Chapter 9.

2. When the Export method of the ICRReport interface is called, your current export property settings are packaged and sent to the Report Engine Server. The packaging is complicated and can fail if the client and server installations of Parallel Crystal are not version compatible. If you experience problems and are sure that the formatting capability is available, get your System Administrator to check the Parallel Crystal installed versions on your client and the Report Server.

Problems with PDF

In Chapter 6, we described how to generate PDF reports using the methods `OutputToPDF`, `ExportToPDF`, `StartParallelPrint` and `ParallelPrintReport`. The `OutputToPDF` method generates PDF directly using the Dynalivery PDF Library. The remaining methods produce PDF indirectly by generating a PostScript version of the report first, and then converting this to PDF using a utility from Adobe called Acrobat Distiller. The problems that arise when generating PDF therefore depend upon which of the methods you have called.

The following problems are known to occur when using the Dynalivery PDF Library with `OutputToPDF`:

1. You cannot generate "linearized" PDF for per-page retrieval to a browser. Future versions of the library may support this feature.

2. OLE objects are rendered as bitmaps. Bitmaps may look grainy when using the zoom feature in Acrobat Reader.

3. Reports with overlapped images may not render correctly.

4. Reports with fonts other than True Type fonts may not render correctly.

In general, we recommend you try to generate PDF using the `OutputToPDF` method first. If problems arise, try `ExportToPDF` that generates PDF using Acrobat Distiller. If problems persist and you have access to Crystal Reports Designer, load your report into the Designer and check for errors in the report itself.

When the Report Engine Server receives a call to `ExportTo`, it saves the name of the PDF output file. Once the report has been generated in PostScript, the Report Engine starts the conversion to PDF by running a Parallel Crystal command line utility called `pdfdistiller`. This program performs the following functions:

- It checks the input PostScript file exists.

- It checks that the output PDF file can be created.

- It runs Adobe Acrobat Distiller in an invisible desktop.

- It prevents multiple versions of Acrobat Distiller from running concurrently.

- It deletes the PostScript input file after conversion.

If you experience problems in generating PDF using `ExportToPDF`, check with your System Administrator that Acrobat Distiller has been installed correctly on your Report Server. The `pdfdistiller` utility searches the registry for the Acrobat Distiller's executable and expects to find an entry in

```
HKLM\Software\Microsoft\Windows\CurrentVersion\AppPaths\AcroDist.exe
```

If the entry no longer points to a valid file, `pdfdistiller` will not be able to execute the `AcroDist` program. If there is no registry entry, then `pdfdistiller` will attempt to run a program called `AcroDist` and will rely on the System PATH environment variable to locate this program.

You can run `pdfdistiller` manually on the Report Server to check its operation. The command line is

```
pdfdistiller [-command ] –input psfile –output pdffile
```

By default, the program runs silently and writes its error messages to the Windows/NT Event Log. Check this log file before you do anything else. If you supply the optional `–command` argument then error messages will be written to the standard output stream as well.

Since Acrobat Distiller relies on the existence of the PostScript file, you should take care to ensure that the file does indeed exist before calling `pdfdistiller`. When you generate PDF via the `ExportToPDF` method, the Report Engine Server applies internal checks to ensure that the PostScript file has been completely generated before returning control to the client. This is because the PostScript file is not generated directly by the Server, but by the Windows Print SubSystem. This process can take several seconds on a busy machine processing large files.

Problems with the Report Server

The following problems can arise with the Report Server:

1. The Load Balancer and Report Retrieval Service can be turned on and off in the Report Server. If you get connection errors when trying to use either of these services, check with the System Administrator that the Report Server is configured to run them.

2. If your client appears to block indefinitely inside a method, then a third party component used by the Crystal Print Engine DLL may have created an error dialog that you are unable to see. To diagnose this problem, you need to get the System Administrator to run Parallel Crystal from the desktop so that all components are visible. If a Crystal API call leads to an error dialog, the Report Engine Server GUI will display the call, and the dialog itself will be visible. The *Parallel Crystal System Administrator's Manual* describes how to run Parallel Crystal in desktop mode.

3. Occasionally components of Parallel Crystal may crash or terminate abnormally. To assist error diagnosis, the Report Engine Server, File Server and Gateway Servers generate log files that contain trace and error messages. In addition components write error messages to the Windows/NT event log. The content of these log files is described in the *Parallel Crystal System Administrator's Manual*.

4. Occasionally a method may fail and report the error:

```
"CORBA Error: CORBA UNKNOWN"
```

   The most likely cause is a Report Engine Server crash rendering the remote method that you are trying to call "unknown".

5. Occasionally a method may fail and report the error:

```
"CORBA Error: CORBA NOIMPLEMENT"
```

   This message means the method you are trying to call cannot be found in the Report Engine Server and the most likely cause is that your C/C++ Client version is inconsistent with the Server version. You should check the installation of Parallel Crystal components on the client and Report Server machines with your System Administrator.

6. A method may fail an report the error:

```
"PCRE API error: Logon failure"
```

   This is a standard Crystal Reports failure message. It occurs if the password and/or username requested by the data source are not supplied correctly. They are normally incorporated into the report file at design time, but you can override the entries when you call the `PELogOnServer` function or the `LogOnServer` method of `PCREEngine`. You should check that the details of the ODBC or other data source administrator on the Report Server match the settings on the machine where the report was designed. To help prevent this problem, we recommend that whenever possible, you design your reports in the same environment in which you expect to run them.

7. A call to the `PrintOut` method may report a `PCREAPIError` with the message:

```
"PCRE API error: Cancelled by user"
```

   This is a standard Crystal Reports failure message that can occur if the report cannot be sent to the nominated printer. You should check the availability of the printer or the print settings established by any calls to the `SelectPrinter` method of `PCREJob`.

Chapter 8
Getting Started with C/C++ Clients

In this chapter, we describe how to customize and generate reports using C or C++. On Windows 95 and Windows/NT systems, you may use any C/C++ development environment capable of loading 32-bit Windows DLLs. You should refer to Chapter 5 for information on how to program a Parallel Crystal ActiveX automation client using Microsoft Visual C++ ActiveX/COM extensions.

This chapter is divided into two major sections

- In the first section entitled Using Parallel Crystal in a C++ Application, we describe how to build a Parallel Crystal client that runs as a C++ application program. The section contains example code fragments that show you how to use C++ classes to connect to the Parallel Crystal Report Server, and how to generate reports formatted as PostScript, PDF and HTML documents.

- In the second section entitled Using Parallel Crystal in a C Application, we describe how to build a Parallel Crystal client that runs as a C application program. The section contains code fragments that show you how to use equivalent C API function calls to connect to the Parallel Crystal Report Server, and how to generate reports formatted as PostScript, PDF and HTML documents.

Both C and C++ applications are constructed using the Parallel Crystal C/C++ Client Distribution Library and its accompanying header files. In C++ environments, the headers provide a class-based interface to this library. In C environments, the headers provide a set of API function prototypes that include the original Seagate Crystal Report Engine functions. Both interfaces are accessed by including the single file, `pcre.h` into your program.

Throughout the Chapter, we use a sample Crystal report file called `Box.rpt` that is distributed with Parallel Crystal in the `SampleReports` directory of the product installation. The file already incorporates its own data so the report may be produced directly without the need to connect to a database.

Before looking at some example code, we need to clarify details of a Parallel Crystal Report Server installation.

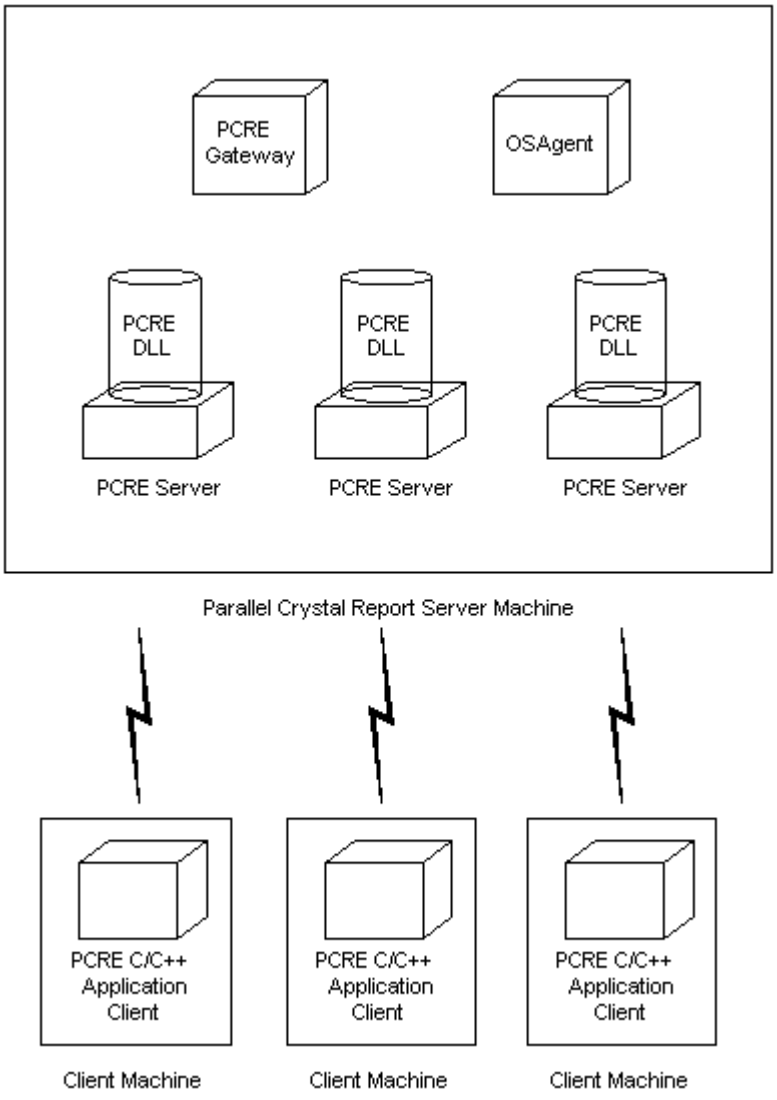


Figure 8.1

Figure 8.1 on page 232 illustrates the following important points:

- The Parallel Crystal Report Server is normally hosted within a single powerful multi-processor server machine running Windows NT.

- A Parallel Crystal C/C++ Application Client normally runs in a less powerful client machine such as a desktop PC running Windows 95.

- Each C/C++ Application Client is connected to a corresponding PCRE Report Engine in the Report Server machine. Each Report Engine includes a copy of the Parallel Crystal Report Engine (PCRE) DLL. Client/server connections are established by an agent on the Report Server called the PCRE Gateway.

- The C/C++ Application Client performs report customization and generation by making calls to the PCRE DLL. These calls are transmitted from the client to the server, and then the results are transmitted back to the client.

- Because report customization is performed within the Report Server, the Crystal report files must be present on that machine, or be available through networked file store.

- Reports are produced on the Report Server machine. However, Parallel Crystal provides a mechanism for retrieving reports back to the client machine.

- When a C/C++ Application Client is started, it is allocated a Report Engine when it connects to the Report Server. The Report Engine is terminated when the client disconnects from the Server.

- Multiple Report Engines execute "in parallel" on the Report Server. However, the extent to which truly concurrent behavior is sustained, depends upon a combination of factors including the number of available processors, the number of connected clients, the overall loading on the machine, and the resulting contention for shared resources.

We're now in a position to look at a simple C++ Application Client that generates a report as a PostScript document.

Using Parallel Crystal in a C++ Application

In this section we describe how to construct a C++ application program that generates a report as a PostScript, PDF or HTML document. For simplicity, we assume that the C++ program has a simple character-based user interface. Once you've mastered the material in this chapter, you can refer to the section entitled Print Job Outputs in Chapter 9 to learn how to generate reports in other formats.

Generating a Simple PostScript Report

The sample C++ program shown in Example 8.1 generates a report from the `Box.rpt` report file and sends it to a PostScript printer. We'll describe it line by line, but note that if you try the program *exactly* as presented here, it will neither compile nor execute.

```
#include <pcre.h>                                        // 1.

void main( int argc, char *argv[] )
{
   PCREAppClient client("example.host.com");             // 2.

   try {
      client.Connect();                                  // 3.
      PCREEngine *engine = client.OpenEngine();          // 4.
      PCREJob *job =                                     // 5.
         engine->OpenJob("C:\\MobileApps\\PCRE\\
                          SampleReports\\Box.rpt");

      job->OutputToPrinter(1);                           // 6.
      job->Start();                                      // 7.
      job->Close();                                      // 8.

      cout << "Report generated" << endl;                // 9.

      engine->Close();                                   // 10.
   } catch ( PCREError& error ) {                        // 11.
      error.Report();                                    // 12.
   }

}
```

Example 8.1

Each numbered paragraph below corresponds to the same numbered code statement

1. Parallel Crystal C++ clients use a Parallel Crystal Client Library called `cppclient.dll`. In order to access the classes in the library, you must include the header file `pcre.h` into your program at compile time, and you must ensure that the installation directories:

   ```
   C:\MobileApps\pcre\CppClient\include
   C:\MobileApps\pcre\Visigenic\vbroker\include
   ```

   are on your include path.

2. The declaration creates an instance of the `PCREAppClient` class. The constructor call specifies the host name of the machine running the Parallel Crystal Report Server as an argument string. The value `"example.host.com"` is fictitious and you must supply a valid host name if you are trying out this code for real.

3. This statement connects the client to the Report Server running on the machine specified in Step 2. It is included in a try-catch block so that if the call to `Connect` fails, and an exception is raised, then control will transfer to the exception handler at statement 11. We'll discuss connection failures in detail in the next chapter, but the most common reasons are that the Parallel Crystal Report Server is not running on the machine identified by the host name, or an incorrect host name was specified.

4. This statement declares a reference to a PCRE print engine. The reference is initialized with the value returned by the call to the `OpenEngine` method of the `client`. In statements 2 through 4, you can see a progression: get a client, connect the client to the Report Server, then open a print engine on the Report Server.

5. The Parallel Crystal Report Engine allows reports to be customized and started by running print jobs. Each job is opened initially by specifying the name of the report file that is to be used to generate the report document. In the example, we specify the file using a full path name for a typical Parallel Crystal installation.

6. Once a print job is opened, it may be customized with calls to the methods of the `PCREJob` class. This statement calls the `OutputToPrinter` method to send the output to a printer. The argument value specifies the number of copies to be printed.

7. The call to the `Start` method generates the report document. When the call returns, the report document will be complete, but printing will not necessarily have finished.

8. When you are finished with a print job, you should call the `Close` method to release resources held in the Report Server.

9. If an error occurs in an API call, the corresponding method of the `PCREEngine` or `PCREJob` class will throw an exception and control will transfer to the exception handler at statement 11. If we reach this print statement, we know the report has been generated.

10. When we are done with all print jobs, the print engine is closed with a call to its `Close` method.

11. The exception handler is executed if an error occurs in any of the preceding method calls. We will discuss error handling in detail in Chapter 9, but for the present, its sufficient to note that by specifying the base class `PCREError` in the handler, we can trap all exception classes raised by the C++ Client.

12. In this simple example, we choose to handle the error by calling its `Report` method that prints the error on the current output stream.

13. The `PCREAppClient` destructor will be automatically invoked on the `client` instance before returning from the function `main`. The destructor will close the client's connection to the Report Server and release resources. If you create the client dynamically on the heap, then you should terminate it with a corresponding call to delete. For example:

```
PCREAppClient *client = new PCREAppClient("example.host.com");
client->Connect();
...
delete client;
```

We'll try to summarize this example as a set of rules that you can use to develop your own applications:

- Access the Parallel Crystal C++ Client Library by including the header file pcre.h.

- Create an instance of the `PCREAppClient` class using the host name (or IP address) of the machine hosting the Report Server.

- Call the `Connect` method to connect your client to a Report Engine running in the Server.

- Open a print engine by calling the `OpenEngine` method of your client.

- Use the `PCREEngine` instance returned by `OpenEngine` to open one or more print jobs. Each job is specified by supplying the full path name of the report file on the Report Server.

- Perform customization and report generation calls for each print job using the methods of the `PCREJob` class.

- Close all your print jobs when they have completed using the `Close` method for each job.

- Close your print engine.

- The `PCREAppClient` destructor terminates your connection to the Report Server. If you declared the client instance dynamically with `new`, remember to destroy it with a corresponding call to `delete`.

- You should not attempt to call delete on instances of the `PCREEngine` and `PCREJob` classes returned by the `OpenEngine` and `OpenJob` methods. Memory for these classes is managed internally by the C++ Client Library.

Trouble Shooting the Simple C++ Client

Let's examine what happens when things go wrong in the simple C++ client.

First, suppose you supplied an incorrect host name in the declaration of the `PCREAppClient` variable. For example, suppose you typed:

```
PCREAppClient client("bad-host name");
```

This statement will be executed correctly, but the subsequent call to `Connect` on line 3 will throw an exception and control will transfer to the handler at line 12. This statement will then print the following line:

```
PCRE Gateway error: Connect: cannot connect client to Gateway
    Service on host "bad-host name".
```

The error message indicates that an attempt to connect the client to the Gateway on the nominated host failed. This usually means one of two things:

- The Report Server host name you supplied in the `PCREAppClient` constructor call was invalid.
- The host name was valid, but Parallel Crystal was not running on the Report Server when the connection was attempted.

A second commonly occurring error is to specify an invalid name for the report file. For example suppose that instead of line 5 you typed:

```
PCREJob *job =
 engine->OpenJob("C:\\MobileApps\\PCRE\\SampleReports\\junk.rpt");
```

When this statement is executed, the report pathname will be supplied to the Print Engine DLL that will fail because the report file does not exist. In this case, the DLL returns an exception to the client that causes control to transfer to the handler at line 12. This statement prints the following line to the standard output stream:

```
"PCRE API error: invalid file name"
```

The "PCRE API error" prefix indicates that the error was raised by the Print Engine DLL. Generally this is an indication that something went wrong in a Print Engine API call, and the most likely cause is an invalid argument value. Unfortunately, the error messages returned by the Print Engine DLL are quite terse. One way to compensate for this is to enclose smaller groups of calls within separate try-catch blocks so that you can be sure which call failed.

A third error that is easy to make is to forget to specify the destination for your report. In other words, once it's generated, where do you send it? In the simple client, line 6 specified that the final report was to be sent directly to a printer rather than be stored in a file. If this line is omitted, then the Print Engine DLL will raise an exception when the client executes the `Start` command on line 7. Control will transfer to the handler on line 12 and the following line will be printed to the standard output stream:

```
"PCRE API error: no print destination specified"
```

This message indicates that the Print Engine DLL could not figure out what to do with your report. You must respond by sending the report to a printer or to a disk file. We'll explore the most common solutions in the next two sections.

Overall, the simple client illustrates the importance of exception handling in Parallel Crystal client programming. In general, you should always ensure that calls to methods in the C++ Client Library are enclosed within an appropriate try-catch block. We'll describe the error handling facilities in the Client Library in more detail in the next chapter.

Generating a Simple PDF Report

Adobe's Portable Document Format (PDF) is widely used as an alternative to PostScript or HTML on the Internet. It provides PostScript quality reproduction (which is far superior to HTML), using a more compact document encoding format.

Example 8.2 illustrates a simple C++ client that outputs a version of the Boxoffice report to PDF. It is in fact identical to Example 8.1 with the exception of the single line highlighted.

```
#include <icstrenm.h>
#include <pcre.h>                                    // 1.

void main( int argc, char *argv[] )
{
   PCREAppClient client("example.host.com");         // 2.

   try {
      client.Connect();                              // 3.
      PCREEngine *engine = client.OpenEngine();      // 4.
      PCREJob *job =                                 // 5.
         Engine->OpenJob("C:\\MobileApps\\PCRE\\
                          SampleReports\\Box.rpt");

      job->OutputToPDF("C:\\MobileApps\\PCRE\\
           SampleOutputs\\Box.pdf", NULL);           // 6.
      job->Start();                                  // 7.
      job->Close();                                  // 8.

      cout << "PDF report generated" << endl;        // 9.

      engine->Close();                               // 10.
   } catch ( PCREError& error ) {                    // 11.
      error.Report();                                // 12.
   }
}
```

Example 8.2

Once again, notice that line 6 is formatted to preserve readability. Splitting a string across physical line boundaries is illegal in C++.

A PDF report is generated by calling the `OutputToPDF` method of the `PCREJob` class and supplying the name of the output file as an argument. The PDF document is generated by starting the print job at Step 7, and is stored in the file `C:\MobileApps\PCRE\SampleOutput\Box.pdf`. In order to display the file, you must have access to a PDF viewer such as Adobe's Acrobat Reader.

The `OutputToPDF` method call uses the Dynalivery PDF Library that extends the Parallel Crystal Report Engine with facilities to render reports directly in PDF. It is possible to provide a set of PDF output options as a second argument to `OutputToPDF`. Further discussion is deferred to Chapter 9. In practice, the default operating options are sufficient for most purposes and are automatically selected when the second argument to `OutputToPDF` is set to `NULL`.

Generating a Simple HTML Report

If the reports you wish to generate are web-based and intended for display exclusively through web browsers, then it may be appropriate to generate the report as an HTML file.

Example 8.3 below illustrates how the Box Office report can be generated in HTML. Again, the code only differs slightly from the preceding examples.

```
#include <pcre.h>                                        // 1.

void main( int argc, char *argv[] )
{
   PCREApplicationClient client("example.host.com");  // 2.

   try {
     client.Connect();                                   // 3.
     PCREEngine *engine = client.OpenEngine();           // 4.
     PCREJob *job =                                      // 5.
            engine->OpenJob("C:\\MobileApps\\PCRE\\
                             SampleReports\\Box.rpt");

      PCREHTMLJobExportOptions options("C:\\MobileApps\\
            PCRE\\SampleOutputs\\Box.html");            // 6.
      job->ExportTo(&options);                           // 6a.

      job->Start();                                      // 7.
      job->Close();                                      // 8.

      cout << "HTML report generated" << endl;           // 9.

      engine->Close();                                   // 10.
   } catch ( PCREError& error ) {                        // 11.
      error.Report();                                    // 12.
   }
}
```

Example 8.3

Generating a report in HTML looks a little more complicated, since two extra statements are required rather than one. In the example, line 6 creates an instance of the `PCREHTMLJobExportOptions` class using the name of the HTML file as an argument to the constructor. This object is passed to the `ExportTo` method that informs the Print Engine DLL that the output for the print job is to be "exported" to HTML. The HTML file is generated when the job is started at line 7. Notice that when a graphical report is generated in HTML, any embedded images will be stored in additional JPEG files in the same directory. If the name of the HTML file is `Box.html`, the JPEG files are named `Box001.jpeg`, `Box002.jpeg`, and so on.

The Parallel Crystal Report Engine supports a large number of additional "export formats" which allow reports to be formatted for a variety of front-office tools. We'll describe these formats in greater detail in Chapter 9.

Summary

- The preceding four sections have described how to construct simple C++ clients that connect to a Parallel Crystal Report Server running on a named host. An exception will be generated if you give an incorrect host name, or Parallel Crystal is not running on that machine.

- Reports are generated by getting access to a print engine, and then opening a print job. When the print job is opened, you supply the full path name of the report file on the Report Server.

- You must tell the Print Engine what to do with the report once it is generated.

- If you want to print a PostScript report, call the `OutputToPrinter` method of the `PCREJob` class.

- If you want to generate a PDF report, call the `OutputToPDF` method and specify the full path name of the PDF file.

- If you want to generate an HTML report, create an instance of a `PCREHTMLJobExportOptions` class and supply the full path name of the HTML file to the constructor. Then call the `ExportTo` method and pass the export-info object as an argument.

- Report output files are not generated until the `StartJob` method is called.

- You must enclose most PCRE method calls within try-catch blocks in order to catch and handle exceptions raised by the C++ Client Library and the Print Engine DLL.

- The `PCREAppClient` destructor automatically closes the connection to the Report Server. If you create an instance of this class dynamically with `new`, then you must have a corresponding call to `delete`.

Using Parallel Crystal in a C Application

When you use Parallel Crystal in a C application, you customize and generate reports using a C-language version of the PCRE API that is, for the most part, identical to the original Seagate Crystal Reports Print Engine interface. The C Client Library automatically transmits the calls across the network to the Report Server, and returns the results to your program.

You need only add two extra function calls to manage your Report Server connection:

```
PEInitialize()
PEUninitialize()
```

In the following sections, we'll demonstrate simple C Application clients that generate reports as PostScript, PDF and HTML documents.

Generating a Simple PostScript Report

The sample C program shown in Example 8.4 generates a report from the `Box.rpt` report file and sends it to a PostScript printer. We'll describe it line by line, but note that if you try the program *exactly* as presented here, it will neither compile nor execute. We'll explain why in the commentary below where each numbered paragraph corresponds to a statement in the code.

```
#include <pcre.h>                                        // 1.
#include <stdio.h>

static void ReportError( short job ) {
   HANDLE handle; short length; char buffer[256];
   PEGetErrorText(job, &handle, &length);                // 11.
   PEGetHandleString(handle, buffer, length)
   printf("%s\n", buffer);
}

void main( int argc, char *argv[] )
{
   short job;
   BOOL bOK = PEInitialize(NULL, "example.host.com",  // 2.
                           argc, argv, PE_DO_CONNECT);
   if ( !bOK ) {
      ReportError(0); goto close_app;                    // 3.
   }

   if ( !PEOpenEngine() ) {                              // 4.
      ReportError(0); goto close_app;
   }
```

Example 8.4 (cont. on next page)

```
   job = PEOpenPrintJob("C:\\MobileApps\\PCRE        // 5.
                        SampleReports\\Box.rpt");
   if ( job == 0 ) {
      ReportError(job); goto close_engine;
   }

   if ( !PEOutputToPrinter(job, 1) ) {              // 6.
      ReportError(job); goto close_job;
   }

   if ( !PEStartPrintJob(job, TRUE) ) {             // 7.
      ReportError(job); goto close_job;
   }

   printf("Report generated\n");

close_job:
   if ( !PECloseJob(job) ) ReportError(job);        // 8.
close_engine:
   PECloseEngine();                                 // 9.
close_app:
   PEUninitialize();                                // 10.

}
```

Example 8.5

Each numbered paragraph below corresponds to the same numbered code statement

1. Parallel Crystal C clients use a Parallel Crystal Client Library called `cppclient.dll`. In order to access the classes in the library, you must include the header file `pcre.h` into your program at compile time, and you must ensure that the installation directories:

   ```
   C:\MobileApps\pcre\CppClient\include
   C:\MobileApps\pcre\Visigenic\vbroker\include
   ```

   are on the include path in your development environment.

2. The call to `PEInitialize` initializes the Client Library and establishes a connection to the Report Server on host `"example.host.com"`.

3. If the function returns `FALSE`, the function `ReportError` retrieves and prints the error message on the standard output stream and control is transferred to Step 10.

4. The call to `PEOpenEngine` opens the print engine in the Report Engine Server. If the function returns `FALSE`, `ReportError` prints the error message and control transfers to Step 10.

5. Once the print engine has been opened, we can customize our report by calling `PEOpenPrintJob` with the full path name of the report file. If the function succeeds, it returns a positive integer that acts as a "handle" for the job. We pass this handle in all subsequent API calls for the job. If the function fails, `ReportError` prints the error message and control is transferred to Step 9.

6. The call to `PEOutputToPrinter` tells the Report Engine to send the report to a PostScript printer when the job is started in Step 7. The second argument specifies the number of copies to print.

7. The call to `PEStartPrintJob` runs the print job and generates the report document. The second argument is required for consistency with the original Seagate API and should always have the value `TRUE`. When the function returns, the report will be generated but it may not necessarily have finished printing.

8. The statement at the label `close_job` closes the print job opened in Step 5. Notice that control is transferred here if preceding API calls made after Step 5 failed.

9. The statement at the label `close_engine` closed the print engine opened in Step 4. Control also transfers here if we failed to open the print job in Step 5.

10. The call to `PEUninitialize` terminates the client's Report Engine Server and disconnects the client from the Report Server machine.

11. The function `ReportError` calls `PEGetErrorText` to get a handle to an error message string, and then calls `PEGetHandleString` to retrieve the message string into a buffer. In practice, these calls access a message cache in the Client Library that stores the error message and code returned by the Report Engine server.

Example 8.4 on page 241, indicates that only two extra statements at steps 2 and 10 are necessary when using the C API to the Parallel Crystal Report Server. The remaining code conforms to the original Seagate Print Engine API.

Trouble Shooting the Simple C Client

Let's examine what happens when things go wrong in the simple C client.

First, suppose you supplied an incorrect host name in the call to `PEInitialize`, and that instead of line 2 you typed:

```
PEInitialize(NULL, "bad-host name", argc, argv, PE_DO_CONNECT);
```

This statement will initialize the Client Library but will fail to connect to the nominated host and will return the value `FALSE`. The call to `ReportError` will then print the following line:

```
PCRE Gateway error: Connect: cannot connect client to Gateway
    Service on host "bad-host name".
```

The error message indicates that an attempt to connect the client to the Gateway on the host called `"bad-host name"` failed. This usually means one of two things:

- The Report Server host name supplied as the second argument to the `PEInitialize` call was invalid.

- The host name was valid, but Parallel Crystal was not running on the Report Server when the connection was attempted.

A second commonly occurring error is to specify an invalid name for the report file. For example suppose that instead of the code in Step 5 in Example 8.4, you typed:

```
job =
  PEOpenPrintJob("C:\\MobileApps\\PCRE\\SampleReports\\junk.rpt");
```

When this statement is executed, the report file pathname will be supplied in a call to the Print Engine DLL and this will subsequently fail because the report file does not exist. In this case, the DLL returns an exception to the client that causes `PEOpenPrintJob` to return a negative result. When `ReportError` is called, it prints the following error message:

```
"invalid file name"
```

Unfortunately, many of the messages returned by the Print Engine DLL are very terse. In some cases it will help if you pass the name of the API function that failed as an extra argument to ReportError. For example:

```
if ( job == 0 ) {
   ReportError("PEOpenPrintJob: ", 0); goto close_engine;
}
```

A third error is not specifying the destination for the report. In other words, once it's generated, where do you send it? In the simple client Example 8.4, Step 6 specified that the final report was to be sent directly to a PostScript printer rather than be stored in a file. If this call is omitted, then the Print Engine DLL will raise an exception when the client executes the `PEStartPrintJob` call on Step 7 and `ReportError` will print the following message:

```
no print destination specified
```

We'll look at other ways to specify the output for a print job in the next two sections.

Generating a Simple PDF Report

Adobe's Portable Document Format (PDF) is widely used as an alternative to PostScript or HTML on the Internet. It provides PostScript quality reproduction, using a more compact document encoding format.

Example 8.5 on page 234 illustrates a simple C client that outputs a version of the Boxoffice report to PDF. We have omitted the `ReportError` function for brevity.

```
#include <pcre.h>                                        // 1.
#include <stdio.h>

void main( int argc, char *argv[] )
{
   short job;
   BOOL bOK = PEInitialize(NULL, "example.host.com",  // 2.
                           argc, argv, PE_DO_CONNECT);
   if ( !bOK ) {
      ReportError(0); goto close_app;                // 3.
   }

   if ( !PEOpenEngine() ) {                          // 4.
      ReportError(0); goto close_app;
   }

   job = PEOpenPrintJob("C:\\MobileApps\\PCRE         // 5.
                        SampleReports\\Box.rpt");
   if ( job == 0 ) {
      ReportError(job); goto close_engine;
   }
   if ( !PEOutputToPDF(job, "C:\\MobileApps\\PCRE     // 6.
         SampleOutputs\\Box.pdf", NULL) ) {
      ReportError(job); goto close_job;
   }
   if ( !PEStartPrintJob(job, TRUE) ) {              // 7.
      ReportError(job); goto close_job;
   }

   printf("Report generated\n");

close_job:
   if ( !PECloseJob(job) ) ReportError(job);         // 8.
close_engine:
   PECloseEngine();                                  // 9.
close_app:
   PEUninitialize();                                 // 10.

}
```

Example 8.6

A PDF report is generated by calling `PEOutputToPDF` and supplying the job handle and the name of the PDF output file as arguments. The PDF document is generated at Step 7 and is stored in the file `C:\MobileApps\PCRE\SampleOutput\Box.pdf` when `PEStartPrintJob` returns. In order to display the file, you must have access to a PDF viewer such as Adobe Acrobat Reader. Most Internet browsers have this viewer pre-installed as plugin.

The `PEOutputToPDF` function uses the Dynalivery PDF Library that extends the Parallel Crystal Report Engine with facilities to render reports directly in PDF. It is possible to provide a set of PDF output options as a third argument to `PEOutputToPDF`, but they require detailed knowledge of PDF and further discussion is deferred to the Chapter 9. In practice, the default operating options are sufficient for most purposes and are automatically selected when the second argument to `PEOutputToPDF` is set to `NULL`.

Generating a Simple HTML Report

If the reports you wish to generate are web-based and intended for display exclusively through web browsers, then it may be appropriate to generate the report as an HTML file.

Example 8.6 below illustrates how the Boxoffice report can be generated in HTML. We have omitted the `ReportError` function for brevity.

```
#include <pcre.h>
#include <stdio.h>
#include <string.h>

void main( int argc, char *argv[] )
{
   PEExportOptions options;                          // 1.
   UXFHTML3Options formatOptions;
   UXDDiskOptions diskOptions;
   short job;

   BOOL bOK = PEInitialize(NULL, "example.host.com",
                           argc, argv, PE_DO_CONNECT);
   if ( !bOK ) {
      ReportError(0); goto close_app;
   }

   if ( !PEOpenEngine() ) {
      ReportError(0); goto close_app;
   }

   job = PEOpenPrintJob("C:\\MobileApps\\PCRE
                        SampleReports\\Box.rpt");
   if ( job == 0 ) {
      ReportError(job); goto close_engine;
   }
```

Example 8.7 (cont. on next page)

```
   formatOptions.fileName =                          // 2.
      "C:\\MobileApps\\PCRE\\SampleOutputs\\Box.html";
   strcpy(options.formatDLLName, "u2fhtml.dll");
   options.formatType = UXFNetscape2Type;
   options.formatOptions = (void *)&formatOptions;
   diskOptions.fileName = "";
   strcpy(options.destinationDLLName, "u2ddisk.dll");
   options.destinationType = UXDDiskType;
   options.destinationOptions = (void *)&diskOptions;

   if ( !PEExportTo(job, &options) ) {               // 3.
      ReportError(job); goto close_job;
   }
   if ( !PEStartPrintJob(job, TRUE) ) {              // 4.
      ReportError(job); goto close_job;
   }

   printf("Report generated\n");

close_job:
   if ( !PECloseJob(job) ) ReportError(job);
close_engine:
   PECloseEngine();
close_app:
   PEUninitialize();

}
```

Example 8.8

The additions required to generate HTML are quite complex and have been highlighted in bold font in the example

1. The three declarations

   ```
   PEExportOptions options;
   UXFHTML3Options formatOptions;
   UXDDiskOptions diskOptions;
   ```

   introduce data structures that are used with the `PEExportTo` call in Step 4. In order to generate an HTML report we must inform the Print Engine that the output of the print job is to be formatted in HTML and stored on disk file. The `formatOptions` and `diskOptions` structures are combined with other information in the variable `options` and this is passed to the Print Engine in Step 4.

2. The first group of statements

```
formatOptions.fileName =
   "C:\\MobileApps\\PCRE\\SampleOutputs\\Box.html";
strcpy(options.formatDLLName, "u2fhtml.dll");
options.formatType = UXFNetscape2Type;
options.formatOptions = (void *)&formatOptions;
```

   enters information into the `options` variable to generate the report in HTML. The member `formatDLLName` is set to the name of a formatting DLL called `"u2fhtml.dll"` and the member `formatType` is set to `UXFNetscape2Type`. `formatOptions` is used as an auxiliary data structure that holds the name of HTML output file, and a pointer to this variable is saved in the `options'` member variable `formatOptions`. The `(void *)` cast is required because options can be used to store details of many other report formats that supply different auxiliary formatting structures.

   The second group of statements

```
diskOptions.fileName = "";
strcpy(options.destinationDLLName, "u2ddisk.dll");
options.destinationType = UXDDiskType;
options.destinationOptions = (void *)&diskOptions;
```

   enters information in `options` to save the report file to disk. The field `destinationDLLName` is set to the name of a destination DLL called `"u2ddisk.dll"` and the field `destinationType` is set to `UXDDiskType`. `diskOptions` is used as an auxiliary data structure that can be used to specify the name of the disk file. However, in this case, the name is already stored in the `formatOptions` structure, so the `fileName` member of `diskOptions` is set to the empty string (`""`).

3. The options data structure initialized in Step 3 is passed to the Print Engine in the call `PEExportTo`. If the parameters are incorrect, the function will return a `FALSE` result and control will transfer to the `close_job` label after the error message has been printed.

The report is generated when the `PEStartPrintJob` is called. If the report contains graphical images, the HTML formatting DLL will store them in additional JPEG files created in the same directory. If the name of the HTML file is `Box.html`, the JPEG files are named `Box001.jpeg`, `Box002.jpeg`, and so on.

The `PEExportTo` call can be used to generate reports in many other formats and dispatch them to disk or to Email servers. We'll describe these facilities in the next chapter.

Chapter 9
Using the C/C++ Client Library

This chapter describes the facilities provided in the C/C++ Client Library. The Parallel Crystal C/C++ Client Library provides a range of report management facilities that you can use with C or C++ applications. The library is accessed through a header file called `pcre.h` that provides a set of C++ class headers or a set of ANSI C function prototypes, depending upon whether you use a C++ or C compiler. The two interfaces are functionally similar.

Error Handling	Console Management	Service Providers
*PCREError PCREFatalError PCREInternalError PCREClientError PCREGatewayError PCREServerError PCREAPIError	*PCREConsole PCREAppConsole	*PCREService PCREServer PCREGateway PCREConfigServer
Client Management	**PCRE API Helpers**	**PCRE API**
*PCREClient PCREAppClient	PCREExportOptions PCREHTMLJobExportOptions PCREPDFJobExportOptions PCREPDFOutputOptions	PCREEngine PCREJob

Figure 9.1

* The C++ classes are arranged in the functional groups shown in Figure 9.1. Within each group, base classes are marked with an asterisk.

The chapter is divided into the following sections

- The section entitled Parallel Crystal Clients describes how to use the `PCREAppClient` class, and in particular, how to specify the connection to your Report Server.

- The section entitled Report Generation for C++ Clients describes how to use the `PCREEngine` and `PCREJob` classes to perform report customization and generation. The methods of the `PCREJob` class are arranged in groups and illustrated with sample code fragments. The section entitled Report Generation for C Clients describes the equivalent facilities for C clients.

- When a report is stored in a file, it may be necessary to retrieve the file from the Report Server back to the machine hosting the client. The section entitled Report Retrieval describes facilities for automatic return of reports to the client.

- The section entitled Error Recovery describes how to use the error handling classes to detect and handle the various errors reported by Parallel Crystal.

- The section entitled Console Interface describes the abstract console interface that error handling classes use to display error messages.

- The section entitled Parallel Crystal Configuration Server describes how clients can access information that describes the way in which their Parallel Crystal installation has been configured.

- The section entitled Parallel Crystal Load Balancer describes how Report Server hosts can be allocated to clients to ensure the load is balanced evenly or systematically across a group of servers.

- The section entitled Multi-Threaded Clients describes how to construct multi-threaded clients that can perform report generation tasks concurrently.

The Parallel Crystal C++ classes do not form a general-purpose class library like the Microsoft Foundation Classes. Their primary purpose is to expose the Report Engine API in a manner that is convenient for C++ programmers, and to provide a small number of additional facilities that assist in integration of report processing into your application.

In the sections that follow, the various facilities provided by the Library are described for both C and C++. For full details of each class, its methods, and its equivalent C interface, you should consult the online Reference Documentation.

Parallel Crystal Clients

In this section, we describe how to create Parallel Crystal clients using C or C++. Each client API uses a set of "service provider" classes to access a corresponding set of Report Server programs that provide report generation, load balancing and connection management services. A client can connect to all servers using a single API call, or can customize individual connections by setting properties through the service providers. In the following sections, we describe how clients are created, how they connect to the Report Server, and how they can customize their connections. We begin with a short overview of the server programs running on the Report Server, and the services they provide.

Services for Clients

When a Parallel Crystal Report Server is started, it makes a range of services available though special-purpose servers:

- The Gateway Server provides an initial point of contact for your client and is responsible for starting a Report Engine Server to generate your report. All clients share access to a single Gateway running on the Report Server. To connect to the Gateway, you must specify the Report Server host and (optionally) the name of the Gateway.

- The Report Engine Server provides report customization and generation facilities. The Report Engine is allocated to your client by the Gateway and is terminated when you disconnect from the Report Server. Report Engine Servers are not shared amongst clients.

- The Configuration Server allows clients to retrieve configuration information about their Parallel Crystal installation on the Report Server. For example, you can obtain the name of the directory containing sample reports or the name of the directory to contain sample output. All clients share access to a single Configuration Server.

- The Load Balancer Server ensures that client connections to multiple Report Servers are allocated in a way that spreads the report processing load evenly across the available server machines. Clients share access to the Load Balancer Server.

- The Custom Server is an extension to the Report Engine Server that is used to provide custom extensions to the Parallel Crystal Report Server.

Parallel Crystal C++ clients gain access to these servers through the `PCREAppClient` class. When you call the client class's `Connect` method, it will establish a connection first to the Gateway server, and then to the allocated Report Engine Server. When you call the `Disconnect` method, these connections are broken and the Report Engine Server is terminated. The `PCREAppClient` class also provides access to client-side "service provider" classes that can be used to fine tune the connection to the Gateway and control the operation of the Report Engine. Access to the Load balancer is granted via special-purpose client-methods. Parallel Crystal C clients gain access to the `PCREAppClient` through an equivalent set of API function calls.

Parallel Crystal client programming always follows the same pattern: you instantiate your client, refine its configuration, and connect to the Report Server. Once you have finished customizing your report, you disconnect from the Report Server. If you use C++, you can rely on the destructor to disconnect your client automatically. If you use C, you must remember to call a disconnect API function. We'll explain these operational details in the following sections.

C++ Application Client Instantiation

The `PCREAppClient` class allows C++ application clients to connect to the Parallel Crystal Report Server. The client has connect- and disconnect- methods and acts as a container for "service provider" classes that provide client-side access to Report Server services. You can install a console in the client to adapt the client's error reporting facilities to the user interface provided by your application.

A `PCREAppClient` is instantiated using one of the following two constructors:

```
PCREAppClient( PCREConsole *console, const char *host = NULL,
               int argc = 0, char * const *argv = NULL );

PCREAppClient( const char *host = NULL, int argc = 0,
               char * const *argv = NULL );
```

The constructor argument `console` specifies an instance of a class implementing the `PCREConsole` interface. If you call the second constructor, then your client will be created with an instance of the default console class `PCREAppConsole`.

The constructor argument `host` specifies the Report Server to which your client will connect. It may take the form of either a symbolic host computer domain name, or a dotted host computer IP address. For example:

```
PCREAppClient("example.host.com");
PCREAppClient("111.222.333.444");
```

If you supply `NULL` or the empty string `""`, or use an abbreviated constructor call

```
PCREAppClient()
```

then your client will be connected to a Report Server somewhere on your network. This is okay if you know that there is only one Report Server and you don't necessarily care about its location. However, if you have multiple Report Servers and you leave the host unspecified, you cannot predict which server the `Connect` call will select. Therefore, you must take steps to ensure that your report files are accessible on all of the Report Servers to which you could possibly connect. This may not be easy so we recommend that you specify a host whenever possible.

The constructor arguments `argc` and `argv` are normally derived from the application's command line arguments, and are used to provide fine control over the connection of the client to the Report Server. In practice, they are only required when connections must be established across the Internet or via a firewall. You should contact our Technical Support staff if you believe you have special operating requirements that will cause difficulty in connecting to your Report Server.

To summarize, here are some typical calls to the PCREAppClient constructor:

```
/*
 * 1. Create an application client for use with a PCRE Report
 *    Server on an unspecified host. A subsequent call to
 *    Connect() will succeed if you have a Parallel Crystal
 *    running somewhere on your network. The client will use
 *    the default character-based PCREAppConsole to report
 *    errors.
 */
PCREAppClient client;

/*
 * 2. Create an application client for use with a PCRE Report
 *    Server on the specified host. A subsequent call to
 *    Connect() will succeed if the host name is valid and
 *    Parallel Crystal is running on that host. The client
 *    will use the default character-based PCREAppConsole
 *    to report errors.
 */
PCREAppClient client("butterfly.acme.com");

/*
 * 3. Create an application client with a user-specified console
 *    that implements the PCREConsole interface and provides an
 *    output console that is integrated with the application's
 *    user interface. The host is unspecified and a call to
 *    Connect() will succeed if Parallel Crystal is running some-
 *    where on your network.
 */
PCREAppClient client(new MyConsole);

/*
 * 4. Combine examples 3 and 4 above. The client is created with
 *    a specified console and Connect() will connect it to the
 *    specified host.
 */
PCREAppClient client(new MyConsole, "ant.acme.com");
```

ANSI C Application Client Initialization

Parallel Crystal C clients are "initialized" rather than "instantiated" using the API function:

```
BOOL PEInitialize( PEConsole *console, const char *host, int argc,
                   char *argv[], int connect );
```

C does not provide default arguments so you must supply a value for each parameter.

The argument console is a pointer to a PEConsole structure that provides access to a PCREConsole implementation. If you supply NULL for this argument, the client will be initialized with a default console that is functionally equivalent to the PCREAppConsole implementation.

The argument `host` specifies the Report Server to which your client will connect. It may take the form of either a symbolic host computer domain name, or a dotted host computer IP address. For example:

```
PEInitialize(NULL, "example.host.com", 0, NULL, PE_DO_CONNECT);
PEInitialize(NULL, "111.222.333.444", 0, NULL, PE_DO_CONNECT);
```

If you supply `NULL` or the empty string `""` for host, then your client will be connected to a Report Server “somewhere" on your network. We recommend that you supply a host name whenever possible, for the reasons stated in the previous section.

The `argc` and `argv` arguments are derived from the command line arguments to your application and are used to provide fine control over the connection process. Again, the remarks in the previous section apply.

The `connect` argument allows you to control whether your client should be initialized and connected in one step, or whether you wish connection to be deferred for some reason. There are three possible values defined in the `pcre.h` header file:

`PE_NO_CONNECT`	Don't connect the client to the Report Server. This value is used if the Load Balancer is to be used in "manual" mode. The connection is made in a subsequent API call.
`PE_DO_CONNECT`	Connect the client to the Report Server. This is the "default" value.
`PE_LB_CONNECT`	Connect the client to the Report Server using the Load Balancer Service in "automatic" mode.

The operation of the Load Balancer is described below in the section entitled Parallel Crystal Load Balancer.

Because C does not support default values for arguments, you must supply a value for each. To save typing, we provide a convenience function called `PEInitRemote` that is defined as

```
BOOL PEInitRemote( const char *host )
{
   return PEInitialize(NULL, host, 0, NULL, PE_DO_CONNECT);
}
```

To summarize, here are some typical calls to the `PEInitRemote` and `PEInitialize` functions:

```
/*
 * 1. Initialize an application client and connect to a
 *    PCRE Report Server on an unspecified host. The client
 *    will use the default character-based PCREAppConsole
 *    to report errors. The call will return FALSE if there
 *    are no Parallel Crystal Report Servers running on your
 *    network.
 */
BOOL bOK = PEInitRemote(NULL);


/*
 * 2. Initialize an application client and connect to a
 *    PCRE Report Server on the specified host. The client
 *    will use the default character-based PCREAppConsole
 *    to report errors. The call will return FALSE if
 *    Parallel Crystal is not running on the specified
 *    host.
 */
BOOL bOK = PEInitRemote("butterfly.acme.com");

/*
 * 3. Initialize an application client with a user-specified
 *    console and connect to a Parallel Crystal Report Server
 *    running on an unspecified host. The call will return
 *    FALSE if there are no Parallel Crystal Report Servers
 *    running on your network.
 */
BOOL bOK = PEInitialize(&myConsole, NULL, 0, NULL, PE_DO_CONNECT);

/*
 * 4. Combine examples 2 and 3 above. The client is initialized
 *    with a user-defined console and connected to a Parallel
 *    Crystal Report Server on the specified host. The call
 *    will return FALSE if Parallel Crystal is not running on
 *    the host.
 */
BOOL bOK = PEInitialize(&myConsole, "ant.acme.com", 0, NULL,
                        PE_DO_CONNECT);
```

C++ Client Configuration

Each `PCREAppClient` instance acts as a container for a number of "service provider" classes that grant clients access to the various Parallel Crystal servers running on the Report Server machine. The service providers are created when you call the client's constructor, and are available through get property methods in the client. When the client calls the `Connect` method, connections are established between each service provider and its corresponding server. Prior to connection, it is possible to configure some service providers by invoking their set-property methods. Frequently, this is used to control operation of the client's Report Engine Server.

The four service providers contained by the client classes are shown in Table 9.1.

Service Provider	Get Property	Purpose (PC = Parallel Crystal)
`PCREGateway`	`getGateway`	Access to PC Gateway Server
`PCREServer`	`getServer`	Access PC Report Engine Server
`PCREConfigServer`	`getConfigServer`	Access to PC Configuration Server
`PCRECustomServer`	`getCustomServer`	Access to PC Custom Server

Table 9.1

The function of each is summarized as follows:

- The `PCREGateway` class provides client access to the Parallel Crystal Gateway Server. The Gateway is responsible for allocating PCRE Report Engine Servers to clients.

- The `PCREServer` class provides client access to the Parallel Crystal Report Engine Server. The Report Engine Server provides the report customization and generation functionality that you use to generate reports.

- The `PCREConfigServer` class provides client access to the Parallel Crystal Configuration Server. The Configuration Server provides information about the current configuration of your Parallel Crystal installation.

- The `PCRECustomServer` class provides special-purpose functionality to clients. Its function may vary depending upon your installation.

The following fragment shows how the get property methods can be used to access the Gateway and Configuration servers.

```
PCREAppClient client;
PCREGateway *gateway = client.getGateway();
PCREConfigServer *config = client.getConfigServer();
```

The `PCREAppClient` class is responsible for managing the life times of its service provider classes. You should not attempt to create a service provider independently, or invoke the `delete` operator on an instance, or access its methods after you have called the client's `Disconnect` or `Quit` methods.

The `PCREGateway` and `PCREServer` providers are described briefly in the next two sections. The `PCREConfigServer` is described in the section entitled Parallel Crystal Configuration Server. If you don't plan to use these "advanced" features of Parallel Crystal, then skip directly to the section entitled Application Client Connection.

Accessing PCREGateway Properties from C++

The `PCREGateway` provides client access to the Parallel Crystal Gateway Server running on the Report Server machine. The property methods allow you to specify the Report Server host and the name of the Gateway running on that host. They also allow you to specify the value of a timeout which controls the period of time that the Gateway will wait while trying to start a Report Engine Server. These properties are summarized in Table 9.2.

Property Name	**Purpose**
`setHost`	Set the Report Server host for the Gateway Server
`getHost`	Return the Report Server host running the Gateway Server
`setName`	Set the name of the Gateway Server
`getName`	Return the name of the Gateway Server
`setWait`	Set Gateway Server timeout when starting a PCRE Server
`getWait`	Return the Gateway Server timeout

Table 9.2

Here is a simple example in which the `setHost` property is used to specify the Report Server machine running the Gateway Server.

```
PCREAppClient client;
try {
   char *hostName = HostNameDialog();
   client.getGateway()->setHost(hostName);
   client.Connect();
   ...
} catch ( PCREError& error ) {
   error.Report();
}
```

It only makes sense to use a set-property before you call the client's `Connect` method. If you call a set-property after connecting the client, the method will throw a `PCREClientError` exception. For example:

```
PCREAppClient client("example3.host.com");
try {
   client.Connect();                      // Connect to "grub".
   char *hostName = HostNameDialog();
   client.getGateway()->setHost(hostName); // Exception thrown!
   ...
} catch ( PCREError& error ) {
   error.Report();
}
```

Accessing PCREServer Properties from C++

The `PCREServer` class provides client access to the Parallel Crystal Report Engine Server running on the Report Server machine. Table 9.3 summarizes the property methods, which allow you to supply or retrieve a string of command line arguments to be used when the Gateway Server starts the Report Engine Server.

Property Name	**Purpose**
`setArguments`	Supply the command line argument string
`getArguments`	Return the command line argument string
`addArgument`	Append a command line argument to the string

Table 9.3

The command line arguments for the Report Engine Server are summarized in Table 9.4.

Argument	**Purpose**
`-verbose`	Switch on Report Engine Server tracing. This is useful if you are running Parallel Crystal on the desktop and you wish to see full tracing in the Report Engine's GUI.
`-folder directory`	Specify the Report Engine Server working directory. This lets you specify report files with path names relative to this directory.
`-client name`	Specify the name of the client. If you are running Parallel Crystal on the desktop, the Report Engine will display this string to help you identify your server amongst many.

Table 9.4

Here are two example fragments that specify the same arguments in equivalent ways:

```
/* Supply arguments in a single call */
try
{
    PCREServer *server = client.getServer();
    server->setArguments(
        "-client toby -folder C:\\MonthlyReports");
    ...
    /* The server will be started with arguments supplied. */
    client.Connect();
}
catch ( PCREError &error )
{
    error.Report();
}
/* Add arguments in multiple calls */
try
{
    PCREServer *server = client.getServer();
    server->addArgument("-client toby");
    server->addArgument("-folder C:\\MonthlyReports");
    ...
    /* The server will be started with arguments supplied. */
    client.Connect();
}
catch ( PCREError &error )
{
    error.Report();
}
```

In each case the argument string is supplied when the Report Engine Server is launched by the Gateway. This implies that you must know the values of the arguments before you issue the `Connect` call. In the section Parallel Crystal Configuration Server we'll show how you can change the Report Engine Server's working folder after you have established the Report Server connection.

C Client Configuration

A Parallel Crystal C client gets access to the service provider classes through "handles" which are returned by the set of API function calls listed in Table 9.5. Each handle carries the value of a read-only opaque type called `PEHANDLE`.

Service Provider Function	Purpose
PEGetGateway	Return a handle to the PCRE Gateway Server
PEGetServer	Return a handle to the PCRE Report Engine Server
PEGetConfigServer	Return a handle to the PCRE Configuration Server
PEGetCustomServer	Return a handle to the PCRE Custom Server

Table 9.5

You must call PEInitialize before calling any of these functions. For example:

```
PEHANDLE gateway;
PEHANDLE config;
BOOL bOK;

bOK = PEInitialize(
         NULL, "example2.host.com", 0, NULL, PE_DO_CONNECT);
gateway = PEGetGateway();
config = PEGetConfigServer();
```

The handles returned by these accessory functions can be used in calls to get or set properties for the associated servers. We'll describe these calls in the next two sections.

Accessing PCREGateway Properties from C

You can access the Gateway Server properties through the API functions listed in Table 9.6. Each function takes the handle returned by a previous call to `PEGetGateway` as its first parameter and gets or sets a property associated with the connection to the Gateway. The functions return the value `TRUE` if they succeed, and `FALSE` if there was an error. The error message can be retrieved with a call to `PEGetErrorText`. If you want to set a property you must call `PEInitialize` with a `PE_NO_CONNECT` argument, then call the set-function, then call `PEConnect` to make the connection using the modified property value. The set-functions return `FALSE` if they are called after `PEConnect`.

Property Name	**Purpose**
`PEGatewaySetHost`	Set the Report Server host for the Gateway Server
`PEGatewayGetHost`	Return the Report Server host running the Gateway Server
`PEGatewaySetName`	Set the name of the Gateway Server
`PEGatewayGetName`	Return the name of the Gateway Server
`PEGatewaySetWait`	Set Gateway Server timeout when starting a PCRE Server
`PEGatewayGetWait`	Return the Gateway Server timeout

Table 9.6

In the following example, the Gateway host and wait period are acquired from user interface dialogs and then set with property calls before completing the connection with a call to `PEConnect`. Notice that you must follow the call to `PEConnect` with a call to `PEInitAPI` to complete the client initialization. The parameter to `PEInitAPI` must be the string `"PCREAPI"`[1].

[1] The calls to `PEConnect` and `PEInitAPI` are only necessary because `PEInitialize` was called with `PE_NO_CONNECT`. Normally, `PEInitialize` makes these calls internally.

```
   PEHANDLE gwHandle;
   char pHost = HostDialog();
   DWORD dwWait = WaitDialog();

   /* Must call PEInitialize with PE_NO_CONNECT.*/
   PEInitialize(NULL, NULL, 0, NULL, PE_NO_CONNECT);

   /* Get Gateway handle and set "host" "wait" properties. */
   gwHandle = PEGetGateway();
   if ( !PEGatewaySetHost(gwHandle, pHost) ) {
      ReportError(); goto close;
   }
   if ( !PEGatewaySetWait(gwHandle, dwWait) ) {
      ReportError(); goto close;
   }

   /* Connect to Report Server on "host". */
   if ( !PEConnect() ) {
      ReportError(); goto close;
   }

   /* Complete initialization. */
   if ( !PEInitAPI("PCREAPI") ) {
      ReportError(); goto close;
   }

   /* Other API calls ... */

close:
   PEUninitialize();
```

The set-functions set properties that affect the client connection to the Report Server. Therefore, it makes no sense to call them after a call to `PEConnect`, so the following code will generate an error message:

```
PEHANDLE gwHandle;
BOOL bOK;

/* Connect to "butterfly" host. */
PEInitRemote("butterfly.acme.com");

/* Get Gateway handle. */
gwHandle = PEGetGateway();

/* Following call generates an error! */
if ( !PEGatewaySetWait(gwHandle, 120) ) {
      ReportError(); goto close;
}
```

Accessing PCREServer Properties from C

You can access the Report Server properties through the API functions listed in Table 9.7. Each function takes the handle returned by a previous call to `PEGetServer` as its first parameter, and gets or sets a command line argument that is to be supplied by the Gateway when its starts the Report Engine Server for the client. The command line argument strings are shown in Table 9.4 on page 258. The functions return the value `TRUE` if they succeed, and `FALSE` if there was an error. The error message can be retrieved with a call to `PEGetErrorText`.

You must call `PEServerSetArgument` and `PEServerAddArgument` after a call to `PEInitialize` with a `PE_NO_CONNECT` argument. Then you must call `PEConnect` and `PEInitAPI` to connect the client to the Report Server. You can call `PEServerGetArguments` before or after a call to `PEConnect`.

Property Name	**Purpose**
`PEServerSetArguments`	Supply the command line argument string
`PEServerGetArguments`	Return the command line argument string
`PEServerAddArgument`	Append a command line argument to the string.

Table 9.7

In the following example, we specify `-client` and `-folder` arguments before connecting to the Report Server.

```
PEHANDLE svHandle;

/* Must call PEInitialize with PE_NO_CONNECT. */
PEInitialize(NULL, NULL, 0, NULL, PE_NO_CONNECT);

/* Get Server handle and set command line arguments. */
svHandle = PEGetServer();
if ( !PEServerSetArguments(
         svHandle, "-client toby -folder C:\\MonthlyReports") ) {
   ReportError(); goto close;
}

/* Connect to Report Server on "host". */
if ( !PEConnect() ) {
   ReportError(); goto close;
}

/* Complete initialization. */
if ( !PEInitAPI("PCREAPI") ) {
   ReportError(); goto close;
}

/* Other API calls ... */

close:
   PEUninitialize();
```

Connecting C++ Clients

When you construct a Parallel Crystal C++ application, you must first create a `PCREAppClient` class, then connect it to the Report Server, and then access the client's Report Engine. Typically, this requires a code sequence such as:

```
PCREAppClient client;
try {
   client.Connect();
   PCREEngine *engine = client.OpenEngine();
   ...
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

When the `Connect` method is called, the following events occur in sequence:

- The client tries to establish a connection to a Gateway running on the Report Server. The Report Server is identified by the host name or host address supplied in the `PCREAppClient` constructor call, or in a call to the `PCREGateway` method `setHost`. In the example no host has been supplied, so the client will attempt to connect to any available Report Server running a Gateway.

- If a connection to the Gateway is established, the client issues a request to the Gateway to start a Report Engine Server on the same host. The Gateway starts the Server with a command line that includes server arguments packaged in the request together with a server name that is unique for the Report Server. If the Server is started within a timeout period (usually 30 seconds), the Gateway returns the host and server name to the client.

- The client establishes a connection to the Report Engine Server using the name and host returned by the Gateway.

Once these connections have been established you can call the client's `OpenEngine` method to open the print engine in the Report Engine Server. The Report Engine is assigned exclusively to your client and cannot be accessed by other clients connected to the same Report Server. These connections last until you call the client's `Disconnect` method or execute its destructor.

If any of the preceding steps fail, the `Connect` method will throw an exception whose type and content identify the nature of the error. A `PCREGatewayError` indicates a failure to connect to the Gateway on the specified host, or a failure to launch the Report Engine Server correctly; a `PCREServerError` indicates a failure to connect to the Report Engine Server. Both exceptions are derived from the `PCREError` class so the easiest way to catch both is to specify `PCREError` in the catch-clause.

Connections to both the Gateway and Report Engine Servers are established using the host and name strings that are supplied through the service provider property methods. The host specifies a particular Report Server on your network, and the name identifies a particular instance of the Server on that host. The connections are established by a component called the OSAgent as follows:

- If both host and name are specified, the OSAgent looks for a named server on the host.
- If only the host is specified, the OSAgent looks for any server on the host.
- If neither host nor name is specified, the OSAgent looks for any server on any host.

The Report Engine Servers are assigned unique names by the Gateway so that they can be identified to their clients. The Gateway Server is normally assigned the default name "pcre". A client is always connected to Gateway and Report Engine Servers on the same host.

Occasionally it may be more appropriate to make connections based on logical Server names rather than Report Server hosts. In order to do this, you must omit the host from the client constructor call and use the `PCREGateway`'s `setName` property method as follows:

```
PCREAppClient client;
try {
   client.getGateway()->setName("purple");
   client.Connect();
   /* Use report server ... */
} catch ( PCREError& e ) { e.Report(); }
```

This code will then try to connect your client to a Report Server running the Gateway named "`purple`". If the connection is made, the Gateway will start a Report Engine Server on the same machine and return its name and host to the client.

You must ensure that the argument supplied to the `setName` method matches the name of a Gateway running on your Report Server. If not, the `Connect` method will throw a `PCREGateway` exception and the connection will fail. In the section entitled Parallel Crystal Configuration Server we'll show how you can retrieve the Gateway name from the Report Server rather than hard-code it as a literal string.

Once a connection has been established, you can retrieve the hosts and names of the Servers as follows:

```
PCREAppClient client;
try {
   client.Connect();
   PCREGateway *gateway = client.getGateway();
   PCREServer *server = client.getServer();
   char *gatewayHost = gateway->getHost();
   char *gatewayName = gateway->getName();
   char *serverHost = server->getHost();
   char *serverName = server->getName();
   /* Use report server ... */
} catch ( PCREError& e ) { e.Report(); }
```

Remember that it makes no sense to try to set the name and host of a Gateway after you have called `Connect`. If you call a set-property in these circumstances, it will throw a `PCREClientError` exception.

Disconnecting C++ Clients

When you are finished with a Parallel Crystal C++ client, you should disconnect it from the Report Server. The easiest way to do this is to invoke the `PCREAppClient` destructor on an instance of the class. For example:

```
void RunReport( char *host, char *report )
{
   PCREAppClient client(host);
   try {
      client.Connect();
      PCREEngine *engine = client.OpenEngine();
      PCREJob *job = engine->OpenJob(report);
      /* API calls ... */
      job->Start();
      engine->Close();
   } catch ( PCREError &e ) {
      throw ReportError(e.Text());
   }
}
```

This function declares an instance of the `PCREAppClient` class as a local variable called `client`. Because the client is instantiated locally, its destructor will be invoked when we leave the function.

It is also possible to disconnect a client manually by calling the `PCREAppClient` class `Disconnect` method. This allows a single instance of `PCREAppClient` to be connected to a given host, then disconnected and re-connected to another host. For example:

```
void RunReport( PCREAppClient *client, char *host, char *report )
{
    try {
      client->getGateway()->setHost(host);
      client->Connect();
      PCREEngine *engine = client->OpenEngine();
      PCREJob *job = engine->OpenJob(report);
      /* API calls ... */
      job->Start();
      engine->Close();
      client->Disconnect();
   } catch ( PCREError &e ) {
      client->Disconnect();
      throw ReportError(e.Text());
   }
}


void main( int argc, char *argv[] )
{
   PCREAppClient *client = new PCREAppClient;
   try {
      RunReport(client, "example.host.com", "C:\\Monthly.rpt");
      RunReport(client, "example2.host.com", "C:\\Summary.rpt");
   } catch ( ReportError& e ) {
      /* report error */
   }
   delete client;
}
```

If your C++ client application crashes due to a program error, the Report Engine Server receives a "client aborted" notification and shuts itself down. However you should not rely on program termination to close your servers and you should endeavor to disconnect your client by calls to `Disconnect` or the `PCREAppClient` destructor.

Connecting C Clients

C clients are automatically connected to the Report Server when you call `PEInitialize` and with a `PE_DO_CONNECT` argument value. For example:

```
PEInitialize(NULL, "example.host.com", 0, NULL, PE_DO_CONNECT);
```

The function returns `TRUE` if the connection is established and `FALSE` otherwise. You must call `PEGetErrorText` to get the reason for the failure. `PEInitialize` is declared with the following signature:

```
BOOL PCRE_API PEInitialize( PEConsole *console, const char *host,
                            int argc, char *argv[], int connect );
```

The parameters have the following meanings:

`console` — Specifies an application console for the client to use. A console provides an abstract interface that insulates the client from the details of your user interface. You seldom need to provide a console in C clients, and if you supply `NULL`, the client will be initialized with a default console that provides character-based output facilities.

`host` — Specifies the domain name or IP address of the Report Server host.

`argc` — Specifies the number of elements in the `argv` vector. Traditionally, it is derived from the value passed to your application's `main` entry function.

`argv` — A null-terminated vector of null-terminated command line argument strings. Traditionally, it is also derived from the value passed to your `main` entry function. `argc` and `argv` are occasionally required to provide special connection parameters to the underlying CORBA networking layer.

`connect` — Specifies whether or not `PEInitialize` should connect your client to the Report Server. The possible values are specified by symbolic constants defined by macros in the header file:

```
#define     PE_NO_CONNECT       0
#define     PE_DO_CONNECT       1
#define     PE_LB_CONNECT       2
```

where `PE_NO_CONNECT` prevents connection, `PE_DO_CONNECT` enables connection, and `PE_LB_CONNECT` enables connection via the Load Balancer. Connection via the Load Balancer is described in the section Parallel Crystal Load Balancer.

Calling the function `PEInitRemote` which is declared with the signature

```
BOOL PCRE_API PEInitRemote( const char *host )
```

is equivalent to calling

```
PEInitialize(NULL, host, 0, NULL, PE_DO_CONNECT);
```

The connection process is exactly the same as that for C++ clients. The client connects to the Gateway server first; then the Gateway allocates a uniquely named Report Engine Server and passes the name back to the client; then the client completes the connection to the Report Engine.

If you want to tailor the connection process in any way, then you must supply a `PE_NO_CONNECT` value to `PEInitialize`. Once you have set the additional connection properties, you must then complete the connection with a call to `PEConnect`, followed by a call to `PEInitAPI`. The following example shows how to connect to a Report Server using the logical name of a Gateway rather than its network host address:

```
PEHANDLE gateway;

/* Must call PEInitialize with PE_NO_CONNECT. */
PEInitialize(NULL, NULL, 0, NULL, PE_NO_CONNECT);

/* Get Gateway handle and specify logical name. */
gateway = PEGetGateway();
if ( !PEGatewaySetName(gateway, "purple") ) {
   ReportError(); goto close; }

/* Connect to Report Server with "purple" Gateway. */
if ( !PEConnect() ) {
   ReportError(); goto close; }


/* Complete initialization. */
if ( !PEInitAPI("PCREAPI") ) {
   ReportError(); goto close; }


/* Other API calls ... */

close:
   PEUninitialize();
```

When your client is connected, you can access a variety of Gateway and Report Engine Server properties via the handles to their service provider classes. For example:

```
PEHANDLE gateway, server;

/* Connect to unspecified Report Server. */
PEInitRemote(NULL);

/* Retrieve gateway details ... */
char host[256], name[256];
gateway = PEGetGateway();
PEGatewayGetName(gateway, name, sizeof name);
PEGatewayGetHost(gateway, host, sizeof host);

/* Retrieve server details ... */
server = PEGetServer();
PEServerGetName(server, name, sizeof name);
PEServerGetHost(server, host, sizeof host);
```

The `PEServerGetXXX` and `PEGatewayGetXXX` functions will return a `FALSE` result if you attempt to call them before the client is connected to the Report Server.

Disconnecting C Clients

You should disconnect your C client from the Report Server by calling `PEUninitialize` before your program terminates. If your client crashes or terminates without calling `PEUnitialize`, the Report Engine Server will receive a "client aborted" signal and shut itself down.

It is possible to re-open a connection to a Report Server by re-calling `PEInitialize` after `PEUninitialize`. This allows you to release resources on the Report Server, if for example, you process a report, then spend a significant amount of time on some other activity, and then process another report.

Report Generation for C++ Clients

In Parallel Crystal, the Report Customization API for C++ clients is exposed through the methods of the `PCREEngine` and `PCREJob` classes. These classes partition the API into two parts: methods that are report engine related and methods that print job related. In this section we'll give a general overview of each class and provide some additional information on how to generate reports in particular formats. To understand the class interfaces in depth, you should consult the documentation supplied online with Parallel Crystal. Assuming installation on drive C, this is located in the directory `C:\MobileApps\pcre\docs` where you should find the following files:

- `CppAPIDocumentation\index.html` is the root index to the PCRE package documentation. The documentation is formatted and cross-referenced in a format similar to the JavaDoc standard.

- `developr.hlp` is the Windows help file for the Seagate Crystal Reports Print Engine API.

You may need to consult both files to understand how the customization function is mapped to an equivalent method of the `PCREEngine` or `PCREJob` classes.

Accessing Report Engines and Print Jobs

Once you have connected your C++ client to the Report Server, you get a reference to the Report Engine by calling the client's `OpenEngine` method:

```
PCREAppClient client;
try {
   client.Connect();
   PCREEngine *engine = client.OpenEngine();
   /* run print jobs */
   ...
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

Your client only has access to a single Report Engine Server so successive calls to `OpenEngine` will return the same value. When you are finished, you should call the `Close` method to release resources in the Report Engine Server. If you call `OpenEngine` when your client is not connected to a Report Server, it will throw a `PCREClientError` exception.

The most important methods of the PCREEngine class are summarized in Table 9.8.

Method	Purpose
OpenJob OpenSubReport PrintReport	Open a print job Open a sub-report Start a report run
LogOnServer LogOffServer	Log on to database server Log off from database server
Open Close	Open a print engine Close print engine

Table 9.8

The PCREEngine class allows you to produce your report in one of three ways

- You can generate the report directly using the PrintReport method. The single argument to the method specifies the name of the report file on the Report Server. No report customization is possible with this method.

- You can open a print job with the OpenJob method. The method takes a single argument specifying the name of the report file on the Report Server, and returns a reference to an instance of the PCREJob class representing the print job. You can call methods of PCREJob to customize the report and then generate it by calling the Start method.

If your report requires access to a database server, then you must call the LogOnServer method before calling OpenJob or PrintReport. NthTableLogon methods allow per table database connections. Since you only have access to a single report engine, it is only possible to open a single database connection at a time. You should call the corresponding LogOffServer method when your report has been generated.

The Close method closes the engine and releases resources within your Report Engine Server. You cannot subsequently call any of the report management methods unless you re-open the engine with another call to Open.

The following code fragment shows some typical calls to the PCREEngine methods:

```
PCREAppClient client;
try {
   PCREServer server = client.getServer();
   client.getServer()->setArguments(
      "-folder C:\\MobileApps\\pcre\\SampleReports");
   client.Connect();
   PCREEngine *engine = client.OpenEngine();

   /* Generate the Box Office report. */
   engine->PrintReport("Box.rpt");

   /* Customize Box Office report. */
   PCREJob *job = engine->OpenJob("Box.rpt");
   job->OutputToPDF
          ("C:\\MobileApps\\pcre\\SampleOutput\\Box.pdf", null);
   job->Start();
   job->Close();

   /* Finished with print engine */
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

The lifetimes of the PCREEngine and PCREJob class instances are managed internally by the PCREAppClient. Consequently their constructors and destructors are private and you should not attempt to create independent instances of these classes or destroy them with delete.

Customizing Print Jobs

The PCREJob class provides methods to perform a wide range of report customization tasks. Each time you call the PCREEngine's OpenJob method, you create a PCREJob instance representing a print job opened on the nominated report file. You can have multiple print jobs opened simultaneously and you may customize and run each job independently of the others. Each call to the PCREJob's Start method will run the job and generate a copy of the report. When you are finished with the print job you should call the PCREJob's Close method to release resources in the Report Engine Server.

In the following sections, we present groups of related PCREJob methods together with sample code fragments illustrating typical calls. For brevity, we omit the surrounding try-catch block, but you must include it in real applications. For more realistic examples, you should consult the sample code distributed with Parallel Crystal.

Print Job Execution

The following group of methods control the execution of a print job and return its status.

```
Start              Close
CloseSubReport
```

For example:

```
PCREJob *job = engine->OpenJob("MyReport.rpt");
/* Customize job ... */
job->Start();
job->Close();
```

After the report has been generated, close the job to release resources in your Report Engine Server. Call `Close` for jobs opened with `OpenPrintJob`. Use `CloseSubReport` for jobs opened with `OpenSubReport`.

Print Job Details

The following group of methods allows you to manipulate details of your report.

```
GetReportTitle    SetReportTitle  SetPrintOptions  HasSavedData
DiscardSavedData  GetPrintDate    SetPrintDate
```

For example:

```
job->SetPrintDate(year, month, day);
job->SetReportTitle("Monthly Sales: January – March");
job->Start();
```

Print Job Outputs

The methods in this group are amongst the most important in `PCREJob` and allow you to specify the format and destination of your report.

```
OutputToPrinter    OutputToPDF    ExportTo
```

We have already given simple examples of calls to OutputToPrinter and OutputToPDF in Chapter 8. In this section we'll show how to use `ExportTo` to generate reports in different output formats and for various storage media.

The `ExportTo` method takes a single parameter that is a pointer to a struct called `PEExportOptions`. This struct encapsulates information about the format and the destination of the report as follows:

```
typedef struct PEExportOptions {
   char formatDLLName[PE_DLL_NAME_LEN];
   DWORD formatType;
   void FAR *formatOptions;
   char destinationDLLName[PE_DLL_NAME_LEN];
   DWORD destinationType;
   void FAR *destinationOptions;
} PEExportOptions;
```

The report format is defined by the `formatDLLName` and `formatType` members. The DLL name specifies a DLL which must be supplied with the Parallel Crystal Installation and which generates the report in the appropriate format. Some formats have minor variations and so the `formatType` name is used to hold a numeric value specifying sub-typing information. For example, there are five variants of Excel format available, represented by constants called `UXFXls2Type`, `UXFXls3Type`, etc. Some output formats take additional format-dependent data, and these are supplied through a variety of helper structs called `UXFxxxOptions`. When a helper struct is supplied, it is referenced through an opaque pointer held in the `formatOptions` field.

The report destination is supplied in a similar way through the `destinationDLLName`, `destinationType` and `destinationOptions` members. The `destinationDLLName` specifies a DLL that is responsible for dispatching the report. The `destinationOptions` member supplies additional destination-dependent data provided through pointers to helper structs called `UXDxxxOptions`. For example, if you specify the `destinationDLLName "u2ddisk"` which saves a report to a disk file, then the `destinationOptions` member should be a pointer to a `UXDDiskOptions` struct that holds the path name of the file to be created.

In Table 9.9 on page 277, we list the possible values for `formatDLLName`, the corresponding sub-type values and the format-dependent helper structs. Then entry "none" means that the `formatOptions` member should be set to `NULL`.

DLL Name	Format Sub-type	Format Options
u2fcr.dll	UXFCrystalReportType	none
u2fdif.dll	UXFDIFType	UXFDIFOptions
u2frec.dll	UXFRecordType	UXFRecordStyleOptions
u2frtf.dll	UXFRichTextFormatType	none
u2fsepv.dll	UXFCommaSeparatedType UXFTabSeparatedType UXFCharSeparatedType	UXFCommaTabSeparatedOptions UXFCommaTabSeparatedOptions UXFCharSepatedOptions
u2ftext.dll	UXFTextType UXFTabbedTextType UXFPaginatedTextType	UXFPaginatedTextOptions
u2fwks.dll	UXFLotusWksType UXFLotusWk1Type UXFLotusWk3Type	none
u2fwordw.dll	UXFWordWinType	none
u2fdoc.dll	UXFWordDosType UXFWordPerfectType	none
u2fqp.dll	UXFQP5Type	none
u2fxls.dll	UXFXls2Type UXFXls3Type UXFXls4Type UXFXls5Type UXFXls5TabType	UXFXlsOptions
u2fhtml.dll	UXFHTML3Type	UXFHTML3Options
u2fmaspdf.dll	UXFPDFDistillerType	UXFPDFOptions

Table 9.9

In the following example, we use the `ExportTo` method to generate a file in Excel format and save it to disk:

```
PCREJob *job = engine->OpenJob("C:\\SampleReports\\Box.rpt");

PEExportOptions options;
UXFXlsOptions formatOptions;
UXDDiskOptions diskOptions;

strcpy(options.formatDLLName, "u2fxls.dll");
options.formatType = UXFXls4Type;
formatOptions.bColumnHeadings = FALSE;
options.formatOptions = (void *)&formatOptions;
strcpy(options.destinationDLLName, "u2ddisk.dll");
options.destinationType = UXDDiskType;
diskOptions.fileName = "C:\\SampleOutputs\\Box.xls";
options.destinationOptions = (void *)&diskOptions;

job->ExportTo(&options);
job>Start();
```

Notice that you must call `Start` after `ExportTo` to actually generate the report.

Table 9.10 below enumerates the possible settings for the `destinationDLLName` and the corresponding values for the `destinationType` and `destinationOptions`:

DLL Name	Destination Sub-type	Destination Options
u2ddisk.dll	UXDDiskType	UXDDiskOptions
u2dmapi.dll	UXDMAPIType	UXDMAPIOptions
u2dpost.dll	UXDExchFolderType	UXDPostFolderOptions

Table 9.10

The following example shows how to mail a report using the MAPI mailer server interface:

```
PCREJob *job = engine->OpenJob("C:\\SampleReports\\Box.rpt");

PEExportOptions options;
UXFXlsOptions formatOptions;
UXDMAPIOptions mailOptions;

strcpy(options.formatDLLName, "u2ftext.dll");
options.formatType = UXFTextType;
options.formatOptions = (void *)NULL;

strcpy(options.destinationDLLName, "u2dmapi.dll");
options.destinationType = UXDMAPIType;
mailOptions.toList = "director@reporthost.com";
mailOptions.ccList = "";
mailOptions.subject = "Box Office Report";
mailOptions.message = "Here is your Box Office Report:\r\n\n";
mailOptions.nRecipients = 0;
options.destinationOptions = (void *)&mailOptions;

job->ExportTo(&options);
job->Start();
```

In order to simplify the specification of output formats and destinations, you can use a helper class called `PCREExportOptions`. This class has constructors that fill in many of the default values for the `PEExportOptions` struct for you, leaving you with the task of invoking property methods to supply the remaining details. For example, you can use a `PCRExportOptions` helper to generate an Excel report as follows:

```
PCREJob *job = engine->OpenJob("C:\\SampleReports\\Box.rpt");

PCREExportOptions options(PCREExportOptions::crEFTExcel40,
                          PCREExportOptions::crEDTDiskFile);
options.setDiskFileName("C:\\SampleOutputs\\Box.xls");

job->ExportTo(&options);
job->Start();
```

Here, the constants `crEFTExcel40` and `crEDTDiskFile` supplied to the constructor allow a `PEExportOptions` struct to be initialized with the correct format and destination values for Excel format and disk storage. You only have to call the `setDiskFileName` property to specify the name of the output file.

The example of emailing a report can also be re-written with similar ease:

```
PCREJob *job = engine->OpenJob("C:\\SampleReports\\Box.rpt");

PCREExportOptions options(PCREExportOptions::crEFTText,
                          PCREExportOptions::crEDTMailMAPI);
options.setMailToList("director@reporthost.com");
options.setMailCCList("");
options.setMailSubject("Box Office Report");
options.setMailMessage("Here is your Box Office Report: \r\n\n");

job->ExportTo(&options);
job->Start();
```

The property methods of the `PCREExportOptions` apply consistency checks to ensure internal consistency of the underlying `PEExportOptions` struct that is built. If these calls generate conflicting values, then the method will throw a `PCREClientError` exception.

It is also possible to use the `PCREExportOptions` helper to generate reports in PDF. When the resulting `PEExportOptions` struct is passed to the Report Engine Server, the export format is noted and then changed internally to PostScript. When the print job is started with a call to the `PCREJob::Start` method, a PostScript report is generated first, and then translated into PDF using Adobe's Acrobat Distiller tool. This provides an alternative method for PDF generation, which can occasionally compensate for deficiencies in the OutputToPDF method described previously. `PCREExportOptions` is used to generate PDF as follows:

```
PCREJob job = engine->OpenJob("C:\\SampleReports\\Box.rpt");

PCREExportOptions options(PCREExportOptions::crEFTPDFDistiller,
                          PCREExportOptions::crEDTDiskFile);
options.setPDFFileName("C:\\SampleOutputs\\Box.pdf");
job->ExportTo(&options);
job->Start();
```

The helper class `PCREPDFJobExportOptions` allows this code to be abbreviated to a single constructor call specifying the name of the PDF output file:

```
PCREPDFJobExportOptions options("C:\\SampleOutputs\\Box.pdf");
job->ExportTo(&options);
job->Start();
```

A similar helper class called `PCREHTMLJobExportOptions` is available for generating reports as HTML documents. This class was used in Example83 above. This class is a specialization of `PCREExportOptions` that automatically sets the formatting DLL to `u2fhtml.dll` and the destination DLL to `u2ddisk.dll`. Note that the version of `u2fhtml.dll` supplied with Parallel Crystal yields performance improvements when rendering image-rich reports on multi-processor Report Servers.

Formula Manipulation

The following group of methods allows you to retrieve and adjust the formula fields and record or group selection formulae in your report.

```
GetNFormulas             GetNthFormula        GetFormula
SetFormula               CheckFormula         GetSelectionFormula
SetSelectionFormula      GetGroupSelectionFormula
SetGroupSelectionFormula CheckGroupSelectionFormula
```

Formula fields are accessed by an index that ranges from `0` to `(GetNFormulas()-1)`. The retrieval methods return a string that is allocated with the Win32 function `GlobalAlloc`. You should release the string with a corresponding call to `GlobalFree`.

The following fragment returns all the formula fields in a report:

```
int nFormulas = job->GetNFormulas();
for ( int i = 0; i < nFormulas; i++ ) {
   char *nameBuffer, *formulaBuffer;
   job->GetNthFormula(i, nameBuffer, formulaBuffer);
   cout << "n = " << i << " name = " << nameBuffer <<
      " formula = " << formulaBuffer << endl;
   job->StrFree(nameBuffer);
   job->StrFree(formulaBuffer);
}
```

The methods `GetFormula` and `SetFormula` allow formula fields to be accessed by name rather than index. The remaining methods have simple get- and set- methods.

Parameter Field and Stored Procedure Manipulation

The following group of methods allows you to manipulate the parameter fields and stored procedure parameters in your report:

```
GetNParameterFields  GetNthParameterField  SetNthParameterField
GetParameterField    SetParameterField
```

Parameter fields are accessed by an index value ranging from `0` to `(GetNParameterFields() – 1)`. The `GetNthParameterField` method accesses the n'th field and returns its attributes in a struct called `PCREParameterFieldInfo`. The following code fragment retrieves and resets all the parameter fields in a report:

```
int nParamFields = job->GetNParameterFields();
for ( int i = 0; i < nParamFields; i++ ) {
   PCREParameterFieldInfo info;
   job->GetNthParameterField(i, &info);
   /* check parameter field ... */
   job->SetNthParameterField(i, &info);
}
```

The `GetParameterField` and `SetParameterField` methods allow slightly more convenient access using the name of the field rather than its index value.

DataBase Table Manipulation

The following group of methods allows you to access database tables within your report:

```
GetNTables           GetNthTableType        GetNthTableLocation
SetNthTableLocation  GetNthTableSessionInfo GetNthTableLogonInfo
SetNthTableLogonInfo TestNthTableConnectivity
```

Tables are accessed by an index value ranging from `0` to `(GetNTables() – 1)`. The following fragment indexes through all the tables in a report and prints the Log on information for each. The method `GetNthTableLogOnInfo` returns information for the n'th table in a struct called `PELogOnInfo`.

```
int nTables = job->GetNTables();
for ( int i = 0; i < nTables; i++ ) {
   PELogOnInfo info;
   job->GetNthTableLogonInfo(i, &info);
   cout << "Table " << i << ": " << info.ServerName << " " <<
      info.DatabaseName << " " << info.UserID << " " <<
      info.Password << endl;
}
```

The methods `GetNthTableSessionInfo` and `GetNthTableLocation` use similar helper structs called `PESessionInfo` and `PETableLocation` to return their results.

SQL Queries

The following group of methods allows you to access the SQL Query string associated with your report:

```
GetSQLQuery    SetSQLQuery
```

The following fragment shows how to retrieve and reset the SQL query:

```
char *query = job->GetSQLQuery();
cout << "SQL Query: " << query << endl;
GlobalFree(query)
job->SetSQLQuery("SELECT * FROM CUSTOMER");
```

Sort Field Manipulation

The following group of methods allows you to retrieve and set the sorting fields for the records and groups within your report:

```
GetNSortFields        GetNthSortField         SetNthSortField
DeleteNthSortField    GetNGroupSortFields     GetNthGroupSortField
SetNthGroupSortField  DeleteNthGroupSortField
```

Sort fields are accessed by an index that ranges from `0` to `(GetNSortFields() – 1)`. The value of the n'th sort field is returned by `GetNthSortField` in a string that you must subsequently release with `GlobalFree`. The following fragment prints all the sort fields in a report:

```
int nFields = job->GetNSortFields();
for ( int i = 0; i < nFields; i++ ) {
   char *name; short direction;
   job->GetNthSortField(i, name, &direction);
   cout << "Sort field " << i << ": " << name << " " <<
           direction << endl;
   GlobalFree(name);
}
```

Sections and Groups

The following group of methods allows you to access the sections and groups within your report:

```
GetNSections    GetNGroups    GetGroupCondition  SetGroupCondition
GetSectionCode
```

Sections are accessed by an index that ranges from `0` to `(GetNSections() – 1)`. The type of a section is retrieved by `GetSectionCode()`, and the condition associated with a group is retrieved as a string accompanied by two scalars identifying the condition and its direction. The following fragment returns the group conditions for all sections in a report:

```
int nSections = job->GetNSections();
for ( int i = 0; i < nSections; i++ ) {
   char *field; short condition, direction;
   int section = job->GetSectionCode(i);
   job->GetGroupCondition(section, field, condition, direction);
   /* Process condition ... */
   GlobalFree(field);
}
```

Area and Section Formatting

The following group of methods allows you to adjust the areas and sections in your report:

```
GetSectionFormat            SetSectionFormat
GetAreaFormat               SetAreaFormat
GetMinimumSectionHeight     SetMinimumSectionHeight
GetSectionFormatFormula     SetSectionFormatFormula
GetAreaFormatFormula        SetAreaFormatFormula
```

Sections and areas are accessed by a section code returned by the `GetSectionCode` method. The formatting details of each section or area are specified in the members of a `PESectionOptions` struct. The following fragment shows how the formatting of the n'th section is retrieved and reset:

```
int section = job->GetSectionCode(n);
PESectionOptions options;
job->GetSectionFormat(section, &options);
/* adjust formatting options ... */
job->SetSectionFormat(section, &options);
```

Subreport Management

The following group of methods allows you to access sub-reports within a given report:

```
OpenSubReport               CloseSubreport
GetNSubreportsInSection     GetNthSubreportInSection
```

Subreports are accessed by `GetNthSubreportInSection`, which returns a `PESubreportInfo` struct containing the name of the sub-report. You can open a job for the sub-report by passing the name to the `OpenSubreport` method of the `PCREEngine` class. The following fragment iterates through all the sub-reports in the n'th section:

```
int section = job->GetSectionCode(n);
int nSubReports = job->GetNSubreportsInSection(section);
for ( int i = 0; i < nSubReports; i++ ) {
   PESubreportInfo info;
   job->GetNthSubreportInSection(section, i, &info);
   PCREJob *subReport = engine->OpenSubreport(job, info.name);
   /* customize sub-report ... */
   subReport->CloseSubreport();
}
```

Graph Manipulation

The following group of methods allows you to manipulate graphs and charts in your report:

```
GetGraphType  GetGraphData  GetGraphText GetGraphOptions
SetGraphType  SetGraphData  SetGraphText SetGraphOptions
```

Graphs are located by a section code returned by the `GetSectionCode` method. Each section can contain multiple graphs that are numbered from 0 upwards. The `GetGraphOptions`, `GetGraphData` and `GetGraphText` methods respectively return the required information in the structs `PEGraphOptions`, `PEGraphDataInfo` and `PEGraphTextInfo`. `GetGraphType` returns an integer-valued type code identifying the type of the graph. The corresponding set methods use the same structs when resetting a property of the graph. The following fragment shows how the formatting options and title string for a graph in the n'th section, are retrieved and reset.

```
int section = job->GetSectionCode(n);
PCREGraphOptions options;
job->GetGraphOptions(section, 0, &options);
options.verticalBars = TRUE;
job->SetGraphOptions(section, 0, &options);
PEGraphTextInfo info;
job->GetGraphText(section, 0, &info);
strcpy(info.graphTitle, "New Title");
job->SetGraphText(section, 0, &info);
```

Report Generation for C Clients

The Report Customization API for C clients is provided through the Parallel Crystal Report Engine Functions accessed through the `pcre.h` header file. These functions are identical to the original Print Engine interface provided with Seagate Crystal Reports and are described in the `developr.hlp` file included in the distribution. Assuming installation on drive C, this is located in the directory `C:\MobileApps\pcre\docs`.

The Customization API for C is summarized in Table 9.11 on page 286. The Print Window Management and Customization functions are not currently available with Parallel Crystal.

Category	**Customization Functions**	
Data Functions	`PEDiscardSavedData`	`PEHasSavedData`
Engine Functions	`PEOpenEngine` `PECanCloseEngine`	`PECloseEngine` `PEGetVersion`
Error Functions	`PEGetErrorCode`	`PEGetErrorText`

Formula Functions	`PEGetFormula`
	`PECheckFormula`
	`PEGetNthFormula`
		`PESetFormula`
		`PEGetNFormulas`

Graph Functions	`PEGetGraphData`
	`PEGetGraphOptions`
	`PEGetGraphText`
	`PEGetGraphType`
		`PESetGraphData`
		`PESetGraphOptions`
		`PESetGraphText`
		`PESetGraphType`

Group Functions	`PEGetNGroups` `PESetGroupCondition`	`PEGetGroupCondition`
String Functions	`PEGetHandleString`	

Parameter Functions	`PEGetNParameterFields`
	`PEGetNthParamInfo`
	`PEConvertPFInfoToVInfo`
		`PEGetNthParameterField`
		`PESetNthParameterField`
		`PEConvertVInfoToPFInfo`

Destination Functions	`PESelectPrinter`
	`PEOutputToPrinter`
	`PEGetExportOptions`
		`PEGetSelectedPrinter`
		`PEOutputToPDF`
		`PEExportTo`

Formatting Functions	PEGetReportTitle PESetReportTitle PEGetPrintOptions PESetPrintOptions PEGetSectionFormat PESetSectionFormat PEGetMargins PESetMargins PESetFont PEGetNDetailCopies PESetNDetailCopies PEGetPrintDate PESetPrintDate PEGetMinimumSectionHeight PESetMinimumSectionHeight
Print Job Functions	PEPrintReport PEGetNPages PEOpenPrintJob PEClosePrintJob PEStartPrintJob PESetJobStatus PECancelPrintJob PEIsPrintJobFinished PEParallelPrintReport PEStartParallelPrintJob
Selection Functions	PEGetSelectionFormula PESetSelectionFormula PECheckSelectionFormula PEGetGroupSelectionFormula PESetGroupSelectionFormula PECheckGroupSelectionFormula
Server Functions	PELogOnServer PELogOffServer PEGetSQLQuery PESetSQLQuery
Sorting Functions	PEGetNSortFields PEGetNthSortField PESetNthSortField PEDeleteNthSortField PEGetNGroupSortFields PEGetNthGroupSortField PESetNthGroupSortField PEDeleteNthGroupSortField
Subreport Functions	PECloseSubreport PEGetNSubreportsInSelection PEGetSubreportInfo PEOpenSubreport PEGetNthSubreportInSection
Stored Procedure Functions	PEGetNParams PEGetNthParam PESetNthParam
Table Management Functions	PEGetNTables PEGetNthTableType PEGetNthtableLogOnInfo PESetNthTableLogOnInfo PEGetNthTableLocation PESetNthTableLocation PETestNthTableConnectivity PEGetNthTableSessionInfo PESetNthTableSessionInfo

Table 9.11

Error Recovery in C Clients

The Parallel Crystal C client has no special facilities for error recovery. Most of the API functions return a boolean result indicating success or failure. If the function fails, you can call `PEErrorCode` and `PEGetErrorText` to get the error code and message returned from the Report Engine Server. Unfortunately, the error text is returned as a "handle" to a string, so an extra call to `PEGetHandleString` is required to retrieve the message as a null-terminated C string. The following code fragment shows a typical function that retrieves the error code and message following an API failure:

```
/*
 * Report the error occurring in the last API call. Job
 * is the print job handle, or 0. The function returns
 * if no error occurred or the message could not be
 * retrieved.
 */
void ReportError( short job ) {
   HANDLE handle; short length, code;
   char buffer[256];

   /* Get last error code. 0 implies no error. */
   if ( !( code = PEGetErrorCode(job) ) ) return;

   /* Retrieve and print last error message. */
   if ( PEGetErrorText(job, &handle, &length) ) {
      if ( length > sizeof buffer ) length = sizeof buffer;
      if ( PEGetHandleString(handle, buffer, length) ) {
         printf("API error %d: %s\n", code, buffer);
      }
   }
}
```

You should call this function after every API function that returns a failure code. For example:

```
if ( !PEOpenPrintJob("C:\\SampleReports\\Box.rpt") ) {
   ReportError(0); goto close_engine;
}
```

Even with the utility function `ReportError` in place, this style of error reporting is tedious and error-prone, and leads to code sequences that are either artificially nested or polluted with goto statements.

Error Recovery in C++ Clients

The Parallel Crystal C++ Client provides a collection of classes that allow API errors to be treated as exceptions. When methods of the `PCREEngine` and `PCREJob` classes detect an error, they throw an exception containing the API error code and message as instance data. You can catch these exceptions by including the method calls within try-catch blocks in your program.

Exceptions are instances of the error handling classes shown in Figure 9.2. All classes are derived from `PCREError`, but `PCREGatewayError`, `PCREServerError` and `PCREAPIError` are additionally derived from `PCREServiceError` in order to group errors that originate in the Report Server.

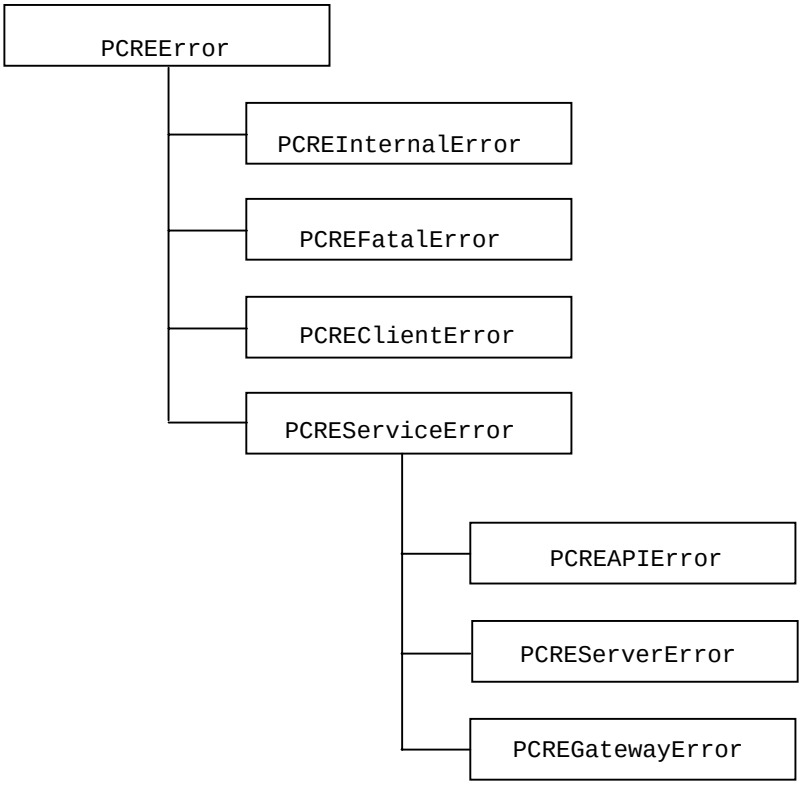


Figure 9.2

The six derived error classes are

- `PCREClientError` is used to identify errors that are detected by the Parallel Crystal C++ client classes. These errors are caused by an invalid argument to a client-side method, or an inappropriate use of such a method.

- `PCREGatewayError` is used to identify errors that are detected in the Gateway Server and are reported back to the client. These errors can arise when the client tries to connect to the Gateway, or when the Gateway tries to start your Report Engine Server.

- `PCREServerError` is used to identify errors that are detected in the Report Engine Server but are not specifically related to the PCRE customization API. For example, they may be generated by a network failure.

- `PCREAPIError` is used to identify errors that are detected by the Crystal Reports Print Engine in the Report Engine Server. The Server transforms the error into an exception containing the Crystal error number and message, and then returns it to the client.

- `PCREFatalError` and `PCREInternal` error are used internally by the C++ client to report catastrophic errors. An internal error usually indicates a logic failure within the code, while a fatal error indicates an external condition that makes it impossible to continue. When handling these errors, you should normally terminate your application in an appropriate way.

In addition to its constructors, each error class has a set of methods for throwing and reporting errors. The `Report` methods use a corresponding method in the `PCREConsole` class to display the error message. So, if you call the `Report` method of `PCREAPIError`, then this method will in turn call the `PCREAPIErrorMsg` method of the client's console.

The hierarchy of error classes allows you to distinguish between various kinds of errors and to handle them individually or as a group. For example, in most of the programming examples we have used a simple try-catch block in the following way:

```
PCREAppClient client("example2.host.com");
try {
   client.Connect();
   PCREOpenEngine *engine = client.Open();
   /* Open job, do API calls ... */
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

Although this code is correct, errors with very different causes will transfer control to the same exception handler, and this may leave your program in an uncertain state. We can be more discriminating in the treatment of errors by using specialized try-catch blocks in a variety of ways:

```
PCREAppClient client("example2.host.com");
try {
   /* Add client customization code here ... */
   client.Connect();
} catch ( PCREError& e ) {                                    // H1.
   /* Handle all connection errors here. */
}

try {
   PCREEngine *engine = client.OpenEngine();
   try {
      PCREJob *job = engine->Open("C:\\ACME\\sales.rpt");

      /* job-processing: phase-1. */
      try {
         /* Customization method calls ... */
      } catch ( PCREAPIError& e1 ) {                          // H2.
         /* Handle API errors in this phase here */
      }

      /* job-processing: phase-2. */
      try {
         /* Customization method calls ... */
      } catch ( PCREAPIError& e2 ) {                          // H3.
         /* Handle API errors in this phase here */
      }

      job->Close();
   } catch ( PCREServiceError& e3 ) {                         // H4.
      /* Handle server errors here */
   }
   engine->Close();
} catch ( PCREError& e ) {                                    // H5.
   /* Handle all remaining errors here */
}
```

Handler H1 handles all possible connection errors. Since the actual errors thrown could be instances of the `PCREGatewayError`, `PCREServerError` or `PCREClientError` classes, the handler uses the `PCREError` base class to trap all three possibilities. You might want to evolve the handler so that a limited number of retries is allowed before terminating the application.

Handlers H2 and H3 handle `PCREAPIErrors` that occur within separate report customization phases. If any other server-side errors occur, they will be handled by the outer handler at H4. This arrangement allows us to proceed with confidence from phase 1 to phase 2. If an API error occurs in phase 1, then the H2 handler can decide whether to allow a retry, or to proceed on some alternative control path.

The handler at H5 catches all errors ignored by the inner handlers. In addition, if we fail to open a print job, we transfer to this point and subsequently call the client's destructor to disconnect from the Report Server.

Table 9.12 shows how each error class's `Report` method calls a corresponding method of the `PCREConsole` interface:

Error Class	Console interface reporting method
`PCREClientError`	Calls `PCREClientErrorMessage`
`PCREGatewayError`	Calls `PCREGatewayErrorMsg`
`PCREServerError`	Calls `PCREServerErrorMsg`
`PCREAPIError`	Calls `PCREAPIErrorMsg`
`PCREFatalError`	Calls `PCREFatalErrorMsg`
`PCREInternalError`	Calls `PCREInternalErrorMsg`

Table 9.12

This association means that whenever you call the `Report` method on an error class (be it a base class like `PCREError`, or a derived class like `PCREAPIError`), the correct `PCREConsole` interface method for the actual error is always called. Of course, you don't need to call the `Report` method at all. If you have trapped the error with an appropriate try-catch block, then how you handle it thereafter is entirely within your control.

Console Interface

The Parallel Crystal C/C++ client library does not make any internal assumptions about the nature of your application. In particular, it does not know whether output messages will be sent to a character stream or displayed in a GUI. Consequently, the reporting facilities incorporated into the error handling classes output error messages using an abstract "application console" interface. You can adapt this interface to your application by providing an appropriate implementation class.

The Parallel Crystal Application Console interface is represented by the abstract base class `PCREConsole`. It contains seven pure-virtual methods that allow a message to be displayed according to the kind of information it conveys. For example, fatal error messages might be accompanied with a sound effect, whereas API errors might simply be written to a text window. You provide the appropriate implementation of these methods in an implementation class derived from `PCREConsole`. If your application uses a simple character-based user interface, then the pre-packaged `PCREAppConsole` class may be adequate for your requirements.

You must install your application console by supplying it as the first argument of the `PCREAppClient` constructor. If you omit the argument, an instance of the character-based `PCREAppConsole` will be installed by default. For example, the fragment

```
PCREAppClient client("example3.host.com");
try {
   client.Connect();
   ...
   try {
      /* API method calls ... */
   } catch ( PCREAPIError& e1 ) { e1.Report(); }
   ...
} catch ( PCREError& e2 ) { e2.Report(); }
```

will report all errors using the default `PCREAppConsole`.

If your application has a graphical user interface, the `PCREAppConsole` class is inappropriate, and you will require a graphical implementation of the `PCREConsole` interface. A typical implementation might take the following form:

```
class GUIConsole: public PCREConsole {
   enum { TEXT_MSG, SERVER_MSG, GATEWAY_MSG,
          CLIENT_MSG, FATAL_MSG, INTERNAL_MSG };
public:
   // PCREConsole interface methods.
   void PCREOutputMsg( const char *msg )
      { MessageBox(TEXT_MSG, msg); }

   void PCREAPIErrorMsg( const PCREError& e )
      { LogMsg(e.getText()); }

   void PCREServerErrorMsg( const PCREError& e )
      { MessageBox(SERVER_MSG, e.getText()); }

   void PCREGatewayErrorMsg( const PCREError& e )
      { MessageBox(GATEWAY_MSG, e.getText()); }

   void PCREClientErrorMsg( const PCREError& e )
      { MessageBox(CLIENT_MSG, e.getText()); }

   void PCREFatalErrorMsg( const PCREError& e )
      { MessageBox(FATAL_MSG, e.getText()); }

   void PCREInternalErrorMsg( const PCREError& e )
      { MessageBox(INTERNAL_MSG, e.getText()); }

private:
   // Implementation.
   void LogMsg( const char *msg );
   void MessageBox( int kind, char *msg );
}
```

The console is installed in the client in the constructor call. The code fragment above then takes the following form:

```
GUIConsole myConsole;
PCREAppClient client(&myConsole, "example3.host.com");
try {
   client.Connect();
   ...
       /* API method calls ... */
   ...
} catch ( PCREError& e ) { e.Report(); }
```

Errors caught by the handler will be reported with the correct `GUIConsole` method since `e` is actually a reference to a derived error class whose `Report` method overrides the base class method. So, if `e` happens to be a `PCREClientError`, it will be reported with the `PCREClientErrorMsg` method of the `GUIConsole`.

Report Retrieval

When a report is stored in a disk file, the file is generated on the Report Server machine using the path name you supplied in an `ExportTo` or `OutputTo` API call. If you wish to display the report, you will have to retrieve the file back to the client machine before invoking an appropriate viewer. The easiest way to do this is to use the Parallel Crystal Report Retrieval Service.

Retrieval to C++ Clients

You activate the Report Retrieval Service for your C++ client by first calling the `setRetrieveMode` property method with the argument value `TRUE`. Then you specify a path for the generated report on the *client's* machine rather than the Report Server machine. It doesn't matter whether you call `setRetrieveMode` before or after you call `Connect`, as long as you call it before the method that specifies the report destination.

The following example shows how the service is used to return a PDF report to the client machine:

```
PCREAppClient client("example.host.com");                    // 1.
try {
   client.setRetrieveMode(TRUE);                             // 2.
   client.Connect();                                         // 3.
   PCREEngine *engine = client.OpenEngine();
   PCREJob *job =
      engine->OpenJob(                                       // 4.
          "C:\\MobileApps\\pcre\\SampleReports\\Box.rpt");
   job->OutputToPDF("C:\\MyReports\\Box.pdf", null);         // 5.
   job->Start();                                             // 6.
   job->Close();
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

Each numbered paragraph below corresponds to the same numbered code statement

1. A `PCREAppClient` is created for connection to a Report Server.

2. The automatic Report Retrieval Service is activated by calling `setRetrieveMode` with the value `TRUE`.

3. The client is connected to the Report Server.

4. A print job is opened for `C:\MobileApps\pcre\SampleReports\Box.rpt`.

5. The report format is set to PDF and the path name `C:\MyReports\Box.pdf` specifies the output file on the client machine.

6. The print job is started. The report is generated on the Report Server machine and is retrieved to the file `C:\MyReports\Box.pdf` on the client machine.

If the report is generated correctly, then the file `C:\MyReports\Box.pdf` will be available on the client machine on return from the `Start` method at Step 6.

The Report Retrieval feature requires your Parallel Crystal to be installed with a file transfer service. If this service is not running then Step 5 above will throw a `PCREServerError` with the message:

```
"PCREServerError: Connect: cannot connect to PCRE File Transfer
 Service"
```

The Report Retrieval feature can be dynamically switched on or off for a client, by calling `setRetrieveMode` with the argument value `TRUE` or `FALSE`.

Retrieval to C Clients

You activate the Report Retrieval Service for your C client by calling the API function `PESetRetrieveMode` with the argument value `TRUE`. You must do this after you call `PEInitialize`. You then specify a path for the generated report on the *client's* machine in one of the API calls `PEOutputToPDF`, `PEOutputToHTML`, `PEStartParallelReport`, `PEStartParallelPrintJob`, or in one of the format options structures supplied in a call to `PEExportTo`. When you call `PEStartPrintJob` or one of the APIs, the report will be generated and then retrieved to the file specified on the client.

The following example shows how the service is used to return a PDF report to the client machine. Some implementation details such as `ReportError` have been ommitted.

```
#include <pcre.h>
#include <stdio.h>

void main( int argc, char argv[] )
{
   short job;
   BOOL bOK = PEInitialize(NULL, "example.host.com",
                           argc, argv, PE_DO_CONNECT);
   if ( !bOK ) {
      ReportError(0); goto close_app; }

   /* Enable report retrieval. */
   PESetRetrieveMode(TRUE);

   if ( !PEOpenEngine() ) {
      ReportError(0); goto close_app; }

   job = PEOpenPrintJob("C:\\MobileApps\\PCRE
                        SampleReports\\Box.rpt");
   if ( job == 0 ) {
      ReportError(job); goto close_engine; }
```

```
   /* Specify output file on client machine. */
   if ( !PEOutputToPDF(job, "C:\\MyReports\\Box.pdf", NULL) ) {
       ReportError(job); goto close_job; }

   if ( !PEStartPrintJob(job, TRUE) ) {
      ReportError(job); goto close_job; }

   printf("Report generated\n");

close_job:
   if ( !PECloseJob(job) ) ReportError(job);
close_engine:
   PECloseEngine();
close_app:
   PEUninitialize();

}
```

Automatic report retrieval is enabled by the call to `PESetRetrieveMode` after the client is initialized. The report is retrieved to the file `C:\MyReports\Box.pdf` when the print job is started by the call to `PEStartPrintJob`.

Parallel Crystal Configuration Server

When Parallel Crystal is installed, a number of directories are created with sample Crystal Reports and demonstration materials. In order to help you locate path names for these directories, Parallel Crystal incorporates a Configuration Server that returns configuration- and installation-dependent information to the client.

Configuration Server for C++ Clients

C++ clients access the Configuration Server through the methods of the `PCREConfigServer` service provider class that are summarized in Table 9.13. Your client is automatically connected to the Configuration Server when you call the `Connect` method. Thereafter you can get a reference to the `PCREConfigServer` class by calling the client's `getConfigServer` method.

Method	**Purpose**
`getPCREVersion`	Returns the Parallel Crystal version string.
`getPCREHostName`	Returns the Parallel Crystal Report Server host name.
`getPCREHostAddress`	Returns the Parallel Crystal Report Server host address.
`getPCREInstallDir`	Returns the Parallel Crystal installation root directory.
`getSampleReportDir`	Returns the Parallel Crystal sample reports directory.
`getSampleOutputDir`	Returns the Parallel Crystal sample output directory.
`getPCREWebServerDir`	Returns the Web Server's root directory.
`getPCREOutputDir`	Returns the Web Server's output directory.
`getPCREGatewayName`	Returns the name of the Parallel Crystal Gateway Server.
`getPCREFileServer`	Returns the file transfer method.
`getPCREConfigInfo`	Returns an aggregate of configuration strings.
`GetPCREDORTempDir`	Returns the DOR temp directory.
`StrFree`	Releases a configuration string.

Table 9.13

Here is a simple example that shows how to retrieve the host name of your Report Server and the version of Parallel Crystal that it is running:

```
PCREAppClient client;
try {
   client.Connect();
   PCREConfigServer *config = client.getConfigServer();
   char *version = config->getPCREVersion();
   char *host = config->getPCREHostName();
   cout << "Connected to PCRE version " << version << " " <<
               host << endl;
   PCREConfigServer::StrFree(version);
   PCREConfigServer::StrFree(host);
} catch ( PCREError& e ) { e.Report(); }
```

To avoid memory leaks, you should release a configuration string when you are finished with it, by calling the static `StrFree` method. If you want to retrieve all configuration strings in a single call, you can use the `getPCREConfigInfo` method that will copy the strings into character arrays contained in a struct called `PCREConfigInfo` that is defined as follows:

```
#define PE_INFO_NAME_LEN   64
#define PE_PATH_NAME_LEN   256

typedef struct PCREConfigInfo {
   char PCREVersion[PE_INFO_NAME_LEN];
   char PCREHostName[PE_INFO_NAME_LEN];
   char PCREHostAddress[PE_INFO_NAME_LEN];
   char PCREInstallDir[PE_PATH_NAME_LEN];
   char PCRESampleReportDir[PE_PATH_NAME_LEN];
   char PCRESampleOutputDir[PE_PATH_NAME_LEN];
   char PCREWebServerDir[PE_PATH_NAME_LEN];
   char PCREOutputDir[PE_PATH_NAME_LEN];
   char PCREGatewayName[PE_INFO_NAME_LEN];
   char PCREFileServer[PE_INFO_NAME_LEN];
} PCREConfigInfo;
```

Notice that you don't need to worry about calling `StrFree` after `getPCREConfigInfo` as the strings have been copied into statically allocated character arrays. For example:

```
PCREAppClient client;
try {
   client.Connect();
   PCREConfigInfo info;
   client.getConfigServer()->getPCREConfigInfo(&info);
   /* use info ... */
} catch ( PCREError& e ) { e.Report(); }
```

The Configuration Server returns information that you can use with other service providers. The next example shows you how to connect to a Report Server and then set the Report Engine Server's working folder to the sample reports directory:

```
PCREAppClient client;
try {
   client.Connect();
   char *samples = client.getConfigServer()->getSampleReportDir();
   client.getServer()->SetFolder(samples);
   PCREConfigServer::StrFree(samples);
   PCREEngine *engine = client.OpenEngine();
   PCREJob *job = engine->OpenJob("Box.rpt");
   ...
   job->Close();
   engine->Close();
} catch ( PCREError& e ) { e.Report(); }
```

The sample report file `Box.rpt` is part of the standard Parallel Crystal installation. However, its full path name is likely to be installation-dependent. We therefore use the `getSampleReportDir` to return the path name of the sample reports directory and then call the `SetFolder` method of the `PCREServer` class to change the working folder of the Report Engine Server accordingly. Thereafter we can refer to all report files by their names relative to the sample directory.

Configuration Server for C Clients

C clients are automatically connected to the Configuration Server when you call `PEConnect` directly, or indirectly through `PEInitialize`. Thereafter, you must call `PEGetConfigServer` to get a handle to the `PCREConfigServer` class, and then call one of the functions listed in Table 9.14. Each function retrieves the configuration string, copies it into a buffer, and returns a value `TRUE`. If the string could not be obtained, the function returns FALSE, and you must call `PEGetErrorText` to determine the reason for failure.

The following simple example shows how to retrieve the host name of the Report Server and the version number of the Parallel Crystal installation:

```
PEHANDLE config;

/* Connect to unspecified report server. */
PEInitRemote(NULL);

/* Obtain configuration info. */
char host[256], version[256];
config = PEGetConfigServer();
PEGetPCREVersion(config, version, sizeof (version));
PEGetPCREHostName(config, host, sizeof (host));

/* Other API calls ... */

/* Disconnect client. */
PEUninitialize();
```

The configuration strings are truncated if the buffer you provide is not large enough.

Method	**Purpose (PC = Parallel Crystal)**
`PEGetPCREVersion`	Returns the PCRE version string
`PEGetPCREHostName`	Returns the PCRE Report Server host name
`PEGetPCREHostAddress`	Returns the PCRE Report Server host address
`PEGetPCREInstallDir`	Returns the PCRE installation root directory
`PEGetPCRESampleReportDir`	Returns the PCRE sample reports directory
`PEGetPCRESampleOutputDir`	Returns the PCRE sample output directory
`PEGetPCREPCREWebServerDir`	Returns the Web Server's root directory
`PEGetPCREGatewayName`	Returns the name of the PCRE Gateway Server
`PEGetPCREConfigInfo`	Returns an aggregate of configuration strings
`PEGetPCREOutputDir`	Returns the PCRE Web Server output directory path
`PEGetPCREDORTempDir`	Returns the PCRE Data Object Reporting temporary path

Table 9.14

The entire collection of configuration strings can be retrieved in a single call to `PEGetPCREConfigInfo`. For example:

```
PCREConfigInfo info;
PEInitialize(NULL, "example2.host.com", 0, NULL, PE_DO_CONNECT);
PEGetPCREConfigInfo(PEGetConfigServer(), &info);
```

Here is another example showing how to use the configuration information to set the Report Server's working folder to the installation's `SampleReports` directory:

```
PCREConfigInfo info;
PEInitRemote("example2.host.com");
PEGetPCREConfigInfo(PEGetConfigServer(), &info);
PEServerSetFolder(PEGetServer(), info.PCRESampleReportDir);
```

Parallel Crystal Load Balancer

When your client has access to a group of Report Servers, you can use the Parallel Crystal Load Balancer service to ensure that the report processing load is distributed appropriately across the available server machines. The load balancer provides two interfaces for automatic and manual use.

When you use automatic mode, the client is connected to a Report Server chosen by the Load Balancer. The choice is made by a selection algorithm, which is normally configured by the System Administrator. The two algorithms most commonly used are called "load factor" and "round robin". The "load factor" algorithm compares all the Report Servers in the group and returns the one with the smallest load. The "round robin" algorithm simply cycles through all Report Servers in turn and selects the next one in the sequence.

When you use manual mode, the Load Balancer returns a vector of Report Server hosts together with their load factors. You use the vector to make your own selection and then reset the host within the client. When you call the client's `Connect` method, you are connected to your chosen Report Server.

The Load Balancer is dynamically updated whenever a Gateway starts or stops a Report Engine Server, when a new Gateway is started, or when an existing Gateway is stopped. This means that it adjusts to servers started by clients using the Load Balancing Service, and to servers started by clients who connected directly to a Report Server without using load balancing.

Using the Load Balancer from C++

C++ clients use the Load Balancer in automatic mode by calling the `Connect` method with the argument `TRUE`. For example:

```
PCREAppClient client;                                  // 1.
try {
   client.Connect(TRUE);                               // 2.
   /* make method calls ... */
} catch ( PCREError& e ) { e.Report(); }               // 3.
```

Each numbered paragraph below corresponds to the same numbered code statement

1. A `PCREAppClient` is declared without specifying a host name. Ultimately, the Load Balancer determines the host

2. `Connect` is called with the value `TRUE` which uses the Load Balancer in automatic mode. The Load Balancer returns the Report Server host determined by its current selection algorithm and the client is connected to that host.

3. If there are no Report Servers available to the Load Balancer, then the `Connect` call will throw a `PCREServerError` exception and at Step 3 the message

   ```
   "Server error: Load Balancer failure: no available Report Servers"
   ```

   will be printed.

To use the Load Balancer in manual mode, you call the `RequestReportServerList` method that returns a vector of candidate hosts. You apply your own selection procedure to the vector and then call the client's `ResetReportServerHost` method passing the selected vector element. Finally, you call `Connect` to connect the client to the Report Server. The following example shows a typical code fragment:

```
PCREAppClient client;
try {
   PCREReportServerInfo servers[100];
   int number=100;
   int nElements;
   nElements = client.RequestReportServerList(servers, number)
   PCREReportServerInfo host = MySelection(servers, number);
   client.ResetReportServerHost(host);
   client.Connect();
   /* method calls ... */
} catch ( PCREError& e.Report(); }
```

Each element of the array returned by `RequestReportServerList` describes a Report Server. The `PCREReportServerInfo structure is defined as follows`:

```
typedef struct PCREReportServerInfo {
   double LoadFactor;
   int NumberOfReportEngineServers;
   int NumberOfReportServerProcessors;
   char GatewayName[PE_REPORT_SERVER_INFO_LEN];
   char ReportServerAddress[PE_REPORT_SERVER_INFO_LEN];
}
```

The Report Server's load factor is given by the value of the `LoadFactor` member and is currently equal to

```
NumberOfReportEngineServers/NumberOfReportServerProcessors
```

The information in the `PCREReportServerInfo` is correct when the call to `RequestReportServerList` returns. However, if you delay making your choice, other clients could connect and render the information obsolete.

If you supply a host in the `PCREAppClient` constructor call, then your client will use the Load Balancer service running on that host. However, the host settings for the Gateway and Configuration Servers are revised automatically whenever the client completes the connection to the selected Report Server.

In order to use the Load Balancer, your Parallel Crystal installation has to be configured to run this service. If the Load Balancer is not running, then calls to `RequestReportServerList` and `Connect` will throw a `PCREServerError` exception with the message:

```
"Server error: cannot connect client to Load Balancer service"
```

Using the Load Balancer from C

C clients use the Load Balancer in automatic mode by supplying `PE_LB_CONNECT` as the last argument to `PEInitialize`. If the Load Balancer is unable to locate a Report Server, `PEInitialize` will return `FALSE` and you should call `PEGetErrorText` to get the associated error message. For example:

```
if ( !PEInitialize(NULL, NULL, 0, NULL, PE_LB_CONNECT) ) {
   HANDLE handle; short length, code;
   if ( PEGetErrorText(0, &handle, &length) ) {
      char msg[256];
      PEGetHandleString(handle, msg, length);
      printf("Client initialization failure: %s\n", msg);
   }
   return;
}
```

The connection process is identical to that used in the C++ client.

To use the Load Balancer in manual mode, you call the `PERequestReportServerInfo` function that returns a vector of candidate hosts. You apply your own selection procedure to the vector and then call the `PEConnectWithReportServerInfo` function passing the selected vector element as argument. The function completes the connection to the Report Server. The following example shows a typical code fragment:

```
PCREReportServerInfo host, servers[100]; int number;
if ( !PEInitialize(NULL, NULL, 0, NULL, PE_NO_CONNECT) ) {
   /* handle error */
}

/* Get a list of available Report Servers. */
number = PERequestReportServerInfo(servers, 100);
if ( number <= 0 ) {
   /* handle error */
}

/* Select the Report Server. */
host = MySelection(servers, number);

/* Connect to selected Report Server. */
if ( !PEConnectWithReportServerInfo(&host) ) {
   /* handle error */
}

/* Make API calls ... */

/* Disconnect from Report Server. */
PEUninitialize();
```

The call to `PERequestReportServerInfo` returns the number of elements in the vector. If this number is less than or equal to zero, an error has occurred and you should call `PEGetErrorText`. The comments at the end of the previous section apply to the C client as well.

Multi-Threaded Clients

The Parallel Crystal C and C++ clients are designed to enable you to build multi-threaded applications. Thread safety is provided using "thread local storage" and synchronized access to C++ class instance variables. The performance impact on the customization methods of the `PCREEngine` and `PCREJob` classes is minimized by managing their instance data with a write-once read-many access protocol.

When you initialize a C or C++ client you create a context that includes the thread, on which the constructor was invoked, the console for the client, the set of service provider classes, and the connection to the Report Server. Since each client has its own Report Engine Server, you can create applications in which multiple client threads can generate reports concurrently.

The following example illustrates a multi-threaded client application in which five clients are concurrently connected to different Report Servers. Each client is created within a worker thread that executes independently of the others. Notice that the lifetime of the thread embraces the lifetime of the client that in turn embraces the lifetime of the Report Engine Server.

```
static const int NTHREADS    = 5;
static char *hosts[NTHREADS] = {
   "beetle", "flea", "mantis", "butterfly", "grub" };

DWORD  WINAPI ClientWorker( LPVOID lpArg )
{
   LPTSTR host = reinterpret_cast<LPTSTR>(lpArg);
   PCREAppClient client(host);
   try {
      client.Connect();
      PCREEngine *engine = client.OpenEngine();
      /* API calls ... */
      engine->Close();
   } catch ( PCREError& e ) {
      e.Report();
      return PCRE_FAIL;
   }

   /* Worker completed. */
   return PCRE_OK;
}


void main( int argc, char *argv[] )
{
   HANDLE handles[NTHREADS];
```

```
   // Create client worker threads.
   for ( int i = 0; i < NTHREADS; i++ ) {
      DWORD dwThreadId;
      handles[i] = CreateThread(NULL, 0, ClientWorker,
                                (LPVOID)hosts[i], 0,
                                &dwThreadId);
      if ( handles[i] == NULL ) {
         cout << "CreateThread: " << GetLastError() << endl;
      }
   }

   // Wait for worker threads to finish.
   WaitForMultipleObjects(NTHREADS, handles, TRUE, INFINITE);

   cout << "All workers terminated" << endl;
}
```

It is possible to adapt this example so that all client threads are connected to Report Servers via the Load Balancer. You simply omit the `host` parameter in the constructor call and call `Connect` with the argument value `TRUE`. It is also possible to have the multiple client threads connect to the same host – possibly to produce different reports. In each case, bear in mind that the actual performance of your client depends upon the number of processors you have available on your client's host. If there is only one, then your threads will execute concurrently but with interleaving in time. In this case, performance would not be improved; it might even be decreased.

A multi-threaded C client can be constructed on very similar lines by adapting the `ClientWorker` function as follows:

```
DWORD  WINAPI ClientWorker( LPVOID lpArg )
{
   LPTSTR host = (LPTSTR)lpArg;
   PEInitialize(NULL, host, 0, NULL, PE_DO_CONNECT);
   PEOpenEngine();
   /* API calls ... */
   PECloseEngine();
   PEUninitialize();
   return PCRE_OK;
}
```

Notice that it is very important to ensure that `PEUninitialize` is called before the ClientWorker function exits. If this is not done, the connection to the Report Server will persist until the application terminates.

A Parallel Crystal C or C++ client is "owned" by the thread which creates the `PCREAppClient` instance or which calls `PEInitialize`. Thereafter, API calls for this client can be made only on the owner thread. For example, the following will fail:

```
DWORD WINAPI ClientWorker( LPVOID lpArg )
{
   PCREAppClient *client =
      reinterpret_cast<PCREAppClient *>(lpArg);
   try {
      client->Connect();
      // Error here – worker thread did not create client!!
      PCREEngine *engine = client->OpenEngine();
      ...
   } catch ( PCREError& e ) {
      e.Report();
      return PCRE_FAIL;
   }

   return PCRE_OK;
}

void main ( int argc, char *argv[] )
{
   PCREAppClient client(argv[0]);
   DWORD dwThreadId;
   HANDLE handle = CreateProcess(NULL, 0, ClientWorker,
                                 (PCREAppClient *)&client,
                                 0, &dwThreadId);
   WaitForSingleObject(handle, INFINITE);
   cout << "worker finished");
}
```

Here the `PCREAppClient` instance is "owned" by the thread executing the constructor. The worker thread created by the call to `CreateProcess` cannot access this client because it does not own the client. You should always structure multi-threaded client applications to ensure that `PCREAppClient` instances are created and accessed within the same thread. Moreover, a single thread can only "own" one client at a time.

Chapter 10
Trouble Shooting C/C++ Clients

This chapter describes how to solve the problems that most frequently occur when developing Parallel Crystal clients in C or C++.

Problems with Client Connections

An attempt to connect your C++ client with the `Connect` method or to connect your C client with the `PEInitialize` or `PEConnect` functions can fail for a number of reasons.

The most common causes are

1. An invalid Report Server host was specified in the client constructor call: either the host name or the IP address is incorrect.

2. Your client was able to connect to the Report Server but the Gateway could not start a Report Engine Server. Perhaps you specified incorrect command line arguments, or the machine was so busy that the Gateway timed out before the Server could get started. Or perhaps the Parallel Crystal Installation has been moved and the Report Server's executable path name (which is configured at install time) is no longer valid. If you experience the timeout problem, use the `setWait` method of the `PCREGateway` service provider to increase the time in seconds. For example, in C++:

```
PCREAppClient client(host);
try {
   // Wait for up to 120 secs for connection.
   client.getGateway()->setWait(120);
   client.Connect();
   /* API calls ... */
 } catch ( PCREError& e ) { e.Report(); }
 client.Quit();
```

   Or in C:

```
PEInitialize(NULL, host, 0, NULL, PE_NO_CONNECT);
PEGatewaySetWait(PEGetGateway(), 120);
if ( !PEConnect() ) { /* report error */ }
if ( !PEInitAPI("PCREAPI") ) { /* report error */ }
/* API calls ... */
PEUninitialize();
```

4. You tried to connect using the Load Balancer but the Load Balancer could not locate any Report Servers. In this case, it is possible that the Report Servers were not configured to use the Load Balancer Service, or there is an inconsistency in the way the Load Balancer Service has been configured. You should get your System Administrator to check the current Load Balancer configuration.

If the problem persists, the causes may be more obscure. First, you should check the status of the OSAgent that administers client/server connections by keeping a dynamic record of the CORBA servers running on your network. The following problems can arise with OSAgents:

5. If there are no OSAgents running on your network, then no client will be able to connect to a Report Server. Your application may hang for around 15 seconds before you get a message from the CORBA implementation indicating that an OSAgent could not be found. Normally you require at least one Report Server to be running the OSAgent.

6. OSAgents communicate with CORBA clients and servers by listening for UDP broadcasts on port 14000. Sometimes its useful to be able to modify this port number – either because its already in use, or to form a separate "domain" in which your client and its Report Server can be tested independently of other production environments. Whenever the OSAgent port is changed, you must supply the new value to your client, and your Report Server must also be re-configured by the System Administrator.

   You can supply the OSAgent port number to your client by setting the `OSAGENT_PORT` environment variable to the required value, or you can use the `ORBagentPort` command line argument and pass this to the client constructor. For example, given:

```
int main(in argc, char *argv[] )
{
   PCREAppClient client("example2.host.com", argc, argv);
   try {
      client.Connect();
       ...
   } catch ( PCREError& e ) { e.Report(); }
}
```

   If the OSAgent is running on port 17000, then you must start the application with the command line:

```
MyApp.exe –ORBagentPort 17000
```

   If your application is a C client, then you can use the same command line and pass the argument in the call to `PEInitialize`:

```
PEInitialize(NULL, host, argc, argv);
```

7. Your client communicates with the OSAgent using UDP broadcast packets that are normally confined to a subnet of your LAN. If you are trying to connect to a Report Server on a remote network, you may need to supply the client with the IP address of the remote OSAgent.

You can supply the OSAgent IP address to your client by setting the `OSAGENT_ADDR` environment variable to the correct value. Alternatively, you can use the command line argument `ORBagentAddr` as follows:

```
MyApp.exe –ORBAgentAddr 123.456.654.321
```

You must collect this argument in the `argv` vector passed to `main`, and pass this in a call to the `PCREAppClient` constructor or `PEInitialize` function, as shown in the preceding example for `ORBagentPort`.

Finally, the following connection problem can arise with badly configured Domain Name Servers:

You may find that connections succeed when specifying a host with an IP address but fail when using a domain name. If this is the case, you should check with your System Administrator that your DNS servers are configured correctly, and can handle both forward and reverse lookups.

Problems with Export Formats

In Chapter 9 in the section entitled Print Job Outputs, we described how to control the output format of the report with a variety of export formats. Two problems can arise with report formats:

1. Not all of the formatting DLLs may be available on your Report Server. If you experience problems with formats other than HTML or PDF, then check with your System Administrator that the corresponding DLL is on your system and accessible to the Report Engine Server through the PATH environment variable.

2. Output formats are set with the `ExportTo` method of the `PCREJob` class. The parameters that this method passes across the network are complicated and require that the versions of the Inprise VisiBroker product installed on the client and Report Server are identical. You should get your System Administrator to check for possible version incompatibility problems.

Problems with PDF

This section describes the problems with PDF production using the Dynalivery library and Acrobat Distiller.

In Chapter 9, we described how to generate PDF reports using the C++ `OutputToPDF` and `ExportTo` methods, and the corresponding API functions in C. The `OutputToPDF` method generates PDF directly using the Dynalivery PDF Library. The `ExportTo` method produces PDF indirectly by generating a PostScript version of the report first, and then converting this to PDF using a utility from Adobe called Acrobat Distiller. The problems that may arise when generating PDF therefore depend upon which of the methods you have called.

The following problems are known to occur when using the Dynalivery PDF Library with `OutputToPDF`:

1. You cannot currently generate "linearized" PDF for per-page retrieval to a browser. Future versions of the library may support this feature.

2. OLE objects are rendered as bitmaps. When using the zoom-in feature in Acrobat Reader, bitmaps may look grainy.

3. Reports with overlapped images may not render correctly.

4. Reports designed with fonts other than True Type fonts may not render correctly.

In general, we recommend you try to generate PDF using the `OutputToPDF` method first. If problems arise, try the `ExportTo` method that generates PDF using Acrobat Distiller. If problems persist and you have access to Crystal Reports Designer, load your report into the Designer and check for errors in the report itself.

When the Report Engine Server receives a call to `ExportTo`, it saves the name of the PDF output file. Once the report has been generated in PostScript, the Report Engine starts the conversion to PDF by running a Parallel Crystal command line utility called `pdfdistiller`. This program performs the following functions:

- It checks the input PostScript file exists.

- It checks that the output PDF file can be created.

- It runs Adobe Acrobat Distiller.

- It prevents multiple instances of Acrobat Distiller from running concurrently.

- It deletes the PostScript input file after conversion.

If you experience problems in generating PDF using `ExportTo`, check with your System Administrator that Acrobat Distiller has been installed correctly on your Report Server. The `pdfdistiller` utility searches the registry for the Acrobat Distiller's executable and expects to find an entry in

```
HKLM\Software\Microsoft\Windows\CurrentVersion\AppPaths\AcroDist.exe
```

If the entry no longer points to a valid file, `pdfdistiller` will not be able to execute the `AcroDist` program. If there is no registry entry, then `pdfdistiller` will attempt to run a program called `AcroDist` and will rely on the System PATH environment variable to locate this program.

You can run `pdfdistiller` manually on the Report Server to check its operation. The command line is

```
pdfdistiller [-command ] –input psfile –output pdffile
```

By default, the program runs silently and writes its error messages to the Windows/NT Event Log. Check this log file before you do anything else. If you supply the optional `command` argument then error messages will be written to the standard output stream as well.

Since Acrobat Distiller relies on the existence of the PostScript file, you should take care to ensure that the file does indeed exist before calling `pdfdistiller`. When you generate PDF via the `ExportTo` method, the Report Engine Server applies internal checks to ensure that the PostScript file has been completely generated before returning control to the client. This is because the PostScript file is not generated directly by the Server, but by the Windows Print SubSystem. This process can take several seconds on a busy machine processing large files.

Problems with the Report Server

The following problems can arise with the Report Server:

1. The Load Balancer and Report Retrieval Service can be turned on and off in the Report Server. If you get connection errors when trying to use either of these services, check with the System Administrator that the Report Server is configured to run them.

2. If your client appears to block indefinitely inside a method of the `PCREEngine` or `PCREJob` classes or in an API function, then a third party component used by the Crystal Print Engine DLL may have created an error dialog on the Report Server. To diagnose this problem, you need to get the System Administrator to run Parallel Crystal from the desktop so that all components are visible. If a Crystal API call leads to an error dialog, the Report Engine Server GUI will display the call, and the dialog itself will be visible. The *Parallel Crystal System Administrator's Manual* describes how to run Parallel Crystal in desktop mode.

3. Occasionally components of Parallel Crystal may crash or terminate abnormally. To assist error diagnosis, the Report Engine Server, File Server and Gateway Servers generate log files that contain trace and error messages. In addition, components write error messages to the Windows/NT event log. The content of these log files is described in the *Parallel Crystal System Administrator's Manual*.

4. Occasionally your client may transfer to a catch-clause and report the error:

   ```
   "CORBA Error: CORBA UNKNOWN"
   ```

   The most likely cause is a Report Engine Server crash rendering the remote method that you are trying to call "unknown".

5. Occasionally your client may transfer to a catch-clause and report the error:

   ```
   "CORBA Error: CORBA NOIMPLEMENT"
   ```

This message means the method you are trying to call cannot be found in the Report Engine Server and the most likely cause is that your C/C++ Client version is inconsistent with the Server version. You should check the installation of Parallel Crystal components on the your client and Report Server machines with your System Administrator.

6. Your client may report a `PCREAPIError` with the message:

```
"PCRE API error: Logon failure"
```

   This is a standard Crystal Reports failure message. It occurs if the password and/or username requested by the data source are not supplied correctly. They are normally incorporated into the report file at design time, but you can override the entries when you call the `PELogOnServer` function or the `LogOnServer` method of `PCREEngine`. You should check that the details of the ODBC or other data source administrator on the Report Server match the settings on the machine where the report was designed. To help prevent this problem, we recommend that whenever possible, you design your reports in the same environment in which you expect to run them.

7. A call to the `Start` method of `PCREJob` may report a `PCREAPIError` with the message:

```
"PCRE API error: Cancelled by user"
```

   This is a standard Crystal Reports failure message that can occur if the report cannot be sent to the nominated printer. You should check the availability of the printer or the print settings established by any calls to the `SelectPrinter` method of `PCREJob`.

Appendix A
Simple ASP/VBScript Tutorial

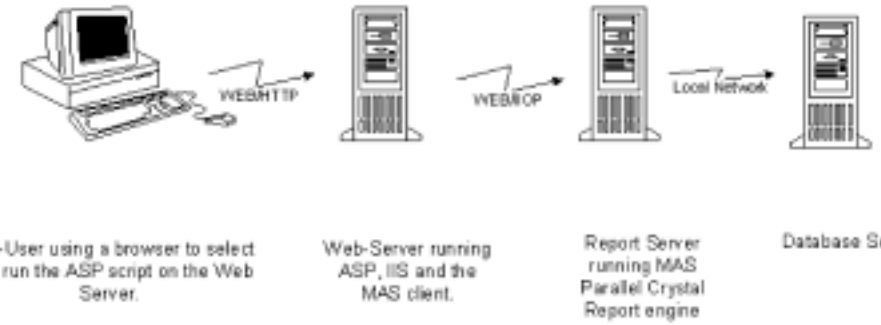
Overview

Note: The files for this tutorial are located in the MobileApps\PCRE\Automation Server\VisualBasic\Tutorial directory.

Additional Note: The MAS Automation Server does not support the Seagate Smart Viewer.

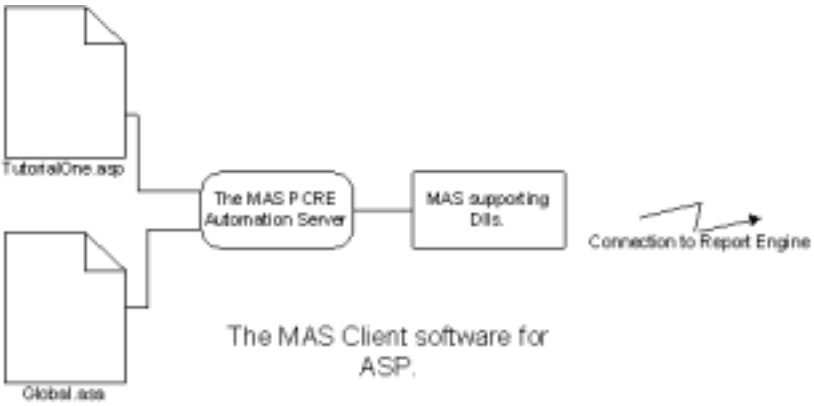
The MAS PCRE product offers a variety of clients that are designed to facilitate report generation in an N-tier (or multiple tier) client-server environment. This document describes how to use one of them, the MAS PCRE Automation Server client, to generate a report under Microsoft Active Server Pages using VBScript.

The context is Active Server Pages 2.0 as supported by IIS 4.0. ASP is an ISAPI application running under IIS, which is designed to construct scripts of considerable power by interfacing to external COM compliant software components - such as the MAS client. A typical reporting system based on ASP will involve two or more computers. Logically, there are four systems involved.


WEB/HTTP
WEB/IOP
Local Network
End-User using a browser to select and run the ASP script on the Web Server.
Web-Server running ASP, IIS and the MAS client.
Report Server running MAS Parallel Crystal Report engine
Database Server

This is the most efficient configuration, with server machines performing the roles for which they are optimised. However, there is no reason, apart from performance, not to run all of these systems in one machine since it can ease development and evaluation.

The software picture on your Web Server looks like this:



The ASP Application
(Script files provided.)

The sequence of events to use an ASP reporting application is

1. The user points his browser at the URL for the script TutorialOne.asp.
2. The script executes and makes the programmed calls on the Report engine.
3. A Report output file is created if all goes well.
4. An HTML response is sent to the user indicating success or failure.
5. The user views the report results.

Understanding a Simple Program

The program is in VBScript and at the end we will show how it embeds into an Active Server pages script. Before we go through it in detail, here is the entire program:

```
On Error Resume Next
Set App = Server.CreateObject("MAS.CRApplication.1")
App.ConnectToReportServer("Tampa")
set Report = App.OpenReport("c:\sampleReport\box.rpt")
Report.ExportToHTML "c:\temp\out.html"
PCREApp.DisconnectFromReportServer()
```

This program is sufficient to cause a report to execute on the designated machine and produce its output in the specified format.

The Automation Server presents an object-oriented interface and to achieve results you are going to have to create at least two objects. These are the Application object and the Report object. VBScript does not require declarations for simple variables, so you will simply see App and Report appear below. The Application object allows you to set up a connection to a report engine on a Report Server.

Start with:

```
On Error Resume Next
```

Should anything go wrong, it can be a little abrupt for the default error handler in ASP to merely give us a "Catastrophic Error" message. The effect of the "On Error Resume Next" statement is to make execution of the script continue after an error. Now if anything goes wrong ,we can see what the specific problem is and report it from an error handling block that we can write.

You create an Application object as follows:

```
Set App = Server.CreateObject("MAS.CRApplication.1")
```

The MAS PCRE product is a true N-tier client server application. Your client knows that the Report engine could be anywhere on the network or could be on the same machine. Unlike the Seagate Automation Server product, there is no need to run the CPU-intensive report engine on your busy Web Server. To make a connection with a report server you tell the Application object what machine it should contact and ask for a report engine on.

```
App.ConnectToReportServer("Tampa")
```

An IP address can be used instead of a machine name. If the report server is not on the local network (it could be half way across the world), then you need to make an additional call to:

```
App.OSAgentAddress().
```

Some reasons the connect call could fail include

- Giving an illegal name.
- The machine might be down.
- There might not be a MAS PCRE product running on that machine.

For more information, see the section below entitled Diagnosing Server Side Problems.

You can check for the success of this (and other commands) like this:

```
If Err.Number = 0 then
Response.write("Connected to server <b>" & "</b><BR>")
Else
       Response.write("<B>Connection to report server error:
</B>")
End If
```

For more information about the response code and how to elicit further diagnostics, see the section below entitled Handling Error information.

The act of connecting to the Report Server has:

- Established a set of CORBA objects on client and server ready to handle any supported API call.
- Gotten the server side PCREGateway to create a personal PCREServer Process.
- Opened a Report Engine in the PCREServer Process.

The next thing to do is to open a report file, which loads that report into the server side report engine. The report file has to be accessible to the server side machine.

```
set Report = App.OpenReport("c:\sampleReport\box.rpt")
<Check for Errors>
```

At this stage we could spend a long time customising the report run on the basis of interactions with the end-user, but the simplest thing to do is just to run the report unchanged.

There are a variety of output formats. To produce HTML, code:

```
Report.ExportToHTML "c:\temp\out.html"
<Check for Errors>
```

Normally, outputs should be directed into the Web Server area, so that the end-user can be given access via an URL to see the results in the browser. ExportToHtml is a MAS invention, which makes a number of API calls to the Report Engine on your behalf to set up an Export structure appropriate for HTML and to run the report. So if there is any reason that the report cannot run successfully on the Report Server, this method call is going to fail. Check that the report is safe by first running it in the Seagate Designer on the report server. The only other common problem is to supply an illegal file name for the HTML, which produces the misleading error message "Disk full" from a system dll.

Finally, we need to close the Report Engine, disconnect from the Report Engine machine, and free the Automation Server objects. This is all done for you in a single call.

```
App.DisconnectFromReportServer()
```

This call:

- Closes any open jobs.
- Closes the Report Engine.
- Shuts down your PCREServer process on the Report Server.
- Closes the CORBA network connection.
- Releases the objects from the report object down that have been created directly or indirectly by your script.

Note that in the nature of COM objects, they may not actually be freed and returned to the heap until the object variables declared in your script go out of scope and the script interpreter also 'releases' them. However, NO use should be made of the objects from report down after the Disconnect call. The connection and the report engine are gone! In order to perform further report processing, you must run the script again or execute another pcreApp.ConnectToReportServer() call which provides you with a new Report Engine and set of objects.

Remember - Parallel Crystal customises and runs a report, but the report must be designed and first tested in the Seagate CRW Designer. If the report does not run in the Seagate Designer, then it is not going to run in Parallel Crystal. The Seagate Designer should be installed on the Report Server, and whether you develop the report there or bring in a functioning report from another machine, the next step is to check that the report runs on the Report Server machine using the designer.

The most common source of problems is the data source not being set up as the report expects on the Report Server. Hit the refresh (lightning) button in the designer to force the report to attempt to use its database settings. If there is a problem, you will get an error message box and you will be in the best position to adjust the report or the machine's data sources and retest.

Embedding the Program in a Working ASP Page

A copy of this ASP script (with error handling) is in the file TutorialOne.asp supplied with Parallel Crystal.

```
<HTML>

<HEAD>

<SCRIPT LANGUAGE="VBScript" RUNAT="Server">

On Error Resume Next
Set App = Server.CreateObject("MAS.CRApplication.1")
App.ConnectToReportServer("Tampa")
set Report = App.OpenReport("c:\sampleReport\box.rpt")
Report.ExportToHTML "c:\temp\out.html"
App.DisconnectFromReportServer()

</SCRIPT>

</HEAD>

<BODY>

</BODY>

</HTML>
```

You also need a global.asa script. Since we are trying to keep things simple in Tutorial One, this file simply says to do nothing at session and application birth and death.

```
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">
Sub Session_OnStart
End Sub
Sub Session_OnEnd
End Sub
Sub Application_OnStart
End Sub
Sub Application_OnEnd
End Sub
</SCRIPT>
```

Finally, let's add basic error handling and tracing, so that we obtain feedback on what happens.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE="VBScript" RUNAT="Server">
sub CheckResponse(comment)
       if Err.Number = 0 then
              Response.write( comment & " succeeded <BR>" )
       Else
              Response.write( "<B>" & comment & " failed <BR></B>"
)
       End If
end sub

On Error Resume Next

response.write("<H3> Tutorial One Application started </H3>")

Set App = Server.CreateObject("MAS.CRApplication.1")

App.ConnectToReportServer("Tampa")
CheckResponse "Connection"

set Report =
App.OpenReport("c:\mobileapps\pcre\sampleReports\box.rpt")
CheckResponse "OpenReport"

Report.ExportToHTML "c:\temp\out.html"
CheckResponse "ExportToHtml"

App.DisconnectFromReportServer()

response.write("Finished <BR>")
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Now all we have to do is place these scripts in a Web Server directory, register them with IIS as an application and hit the tutorialOne.asp URL with a browser to test it.

Setting Up Your First ASP Application

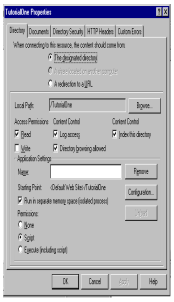
An ASP Web Application is a collection of scripts and possibly HTML files and ActiveX components which live in a directory on the Web Server. This directory must be identified to the Web Server in order to "create" the application.

Go to your Web Server root (usually something like c:\inetpub\wwwroot) and create a tutorialOne folder. Place in it the global.asa and the tutorialOne.asp files supplied.

- Start the MMC. (Microsoft Management Console)

Program/
Windows NT 4.0 Options Pack/
Microsoft Internet Information Server\
Internet Service manager

- Select the directory.
- Right click and select Properties.
- Fill in the form (see the example below). Select the Isolate Process check box and click OK



Note of Explanation: *If you do not tick the "Isolate Process" box, then the MAS client is run in the main IIS process. You can see this process in the Task Manager. It is called InetInfo.exe. Should there be any kind of program error in this process, then the whole Web Server is brought down. There is no need to take this risk. Select Isolate Process and your application will be run in a separate proxy process called MTS.exe. This is designed that should the application crash, then the next user to connect simply gets a new copy of the application.*

In the Event of a Failure

In the event of a failure, you are going to see some large numbers coming back from the Automation Server as error numbers. These are standard COM error codes. Failure codes in COM lead off with the top bit set, so they appear as very large negative numbers. Always print them in hexadecimal form and then you can look them up in <winerror.h> in extremis. Normally, these are one of two generic COM errors to let you know something is wrong, and then we provide a means to obtain further information on the problem.

The two COM codes that you will see are:

```
E_FAIL       0x80004005
E_UNEXPECTED 0x8000FFFF
```

You will be able to get a much more informative error code and error description string from the product. This information can be obtained by using the .LastErrorCode and .LastErrorString methods in the Application object and in the Report object or by looking at the Server Log. The Server Log is a very useful list of all the commands that have been accepted by the Report Engine process on the server. Its name is server.log and it lives in mobileapps/pcre/server.log. In the server.log, following a command that has failed, you will normally find an explanation of why it failed.

Here is an extract from a Server Log:

```
… OpenPrintJob File:
"c:\mobileapps\pcre\SampleReports\rubbish.RPT"
… CREAPI Error:     returned for job: 0
… Error text:       Invalid file name.
```

It can be helpful in development to run the PCRE product in foreground instead of as an NT service. The log entries for PCREGateway and PCREserver processes are then visible in GUI form on the Report Server's screen. To do this, stop the Parallel Crystal service and then execute osagent.exe and pcregateway,exe.

Diagnosing Server-side Setup Problems

Connection. You are unable to get started. The app.Connect("machine") is failing.

- Check that the machine name or IP number is correct.
- Check that the machine is up.
- Check that the machine can be accessed with Ping or Network Neighborhood.
- Check that the MAS PCRE product is running on the machine.
- If running as a service, look in Settings/Control Panel/Services to check that the NT service "Parallel Crystal" is started.
- Check that OsAgent and PCREGateway are active by using the Task Manager Process List.
- Open a DOS box and type PCREServer. Check that a PCREServer GUI appears before dismissing it.

Diagnosing Client (Web Server)-side Setup Problems

The simplest way to set up correctly on the Web Server is by using a standard MAS installation to put the whole product or the relevant client part on that machine. If you need to install or check the MAS client manually, there are only three files involved at the time of writing.

Cpemas1.dll .....The Automation Server

Cppclient.dll.....The supporting C/C++ client DLL

Orb_r.dll.....The CORBA Orb Dll handling network communications

You must ensure that the cppclient.dll and orb_r.dll are in directories which are mentioned in the PATH environment variable and that there are no earlier versions of these files in directories mentioned in PATH. Normally, they are placed in a directory called c:/mobileapps/pcre/bin.

You must register the cpemas1.dll. To register the dll, open a Command Prompt and type "regsvr32 cpemas1.dll". A message box will confirm if the registration was successful. Cpemas1.dll is a COM object and can be examined with various Microsoft tools.